Università degli Studi della Calabria

Dipartimento di Matematica

Dottorato di Ricerca in Matematica ed Informatica

XXII Ciclo

Settore Disciplinare INF/01 INFORMATICA

Tesi di Dottorato

Towards grounding Semantic Web to Answer Set Programming

Alessandra Martello

A.A. 2008-2009

Supervisore
Prof. Giovambattista Ianni

Coordinatore
Prof. Nicola Leone

Towards grounding Semantic Web to Answer Set Programming

Ph.D. Thesis

Alessandra Martello

December 2, 2009

Author's Address:
Alessandra Martello
Dipartimento di Matematica, Cubo 30B
Università degli Studi della Calabria
Via Pietro Bucci
I-87036 Rende, Italy, EU
e-mail: `martello@mat.unical.it`

To my family

Contents

Introduction

This thesis focuses on the problem of dealing with Semantic Web data in a new way. In particular, it is taken in consideration the opportunity of exploiting the Answer Set Programming (ASP) methodology to meet many of the requirements the community is still investigating on. Besides suggesting a possible way of solving known issues, we also provide higher capabilities to the overall picture of the Semantic Web .
The goal of this thesis is tackled from a twofold perspective: on the one hand, we consider the practical feasibility of adopting ASP technologies for actual Semantic Web applications. In particular, we look at ASP as an appropriate technology for building the necessary infrastructure for managing, storing, querying and reasoning over Semantic Web data, with a special focus on RDF information sources. On the other hand, we formally show how ASP can deal with some open knowledge representation issues: in this respect we will illustrate the feasibility of ASP as a modeling language for representing an integrating knowledge sources, and define their semantics.

Motivation of the work and objectives

Motivation. Thanks to the exciting promises of the Semantic Web vision, a lot of research has been carried out about that topic [56], allowing for several infrastructure components of this vision to be fixed and suggesting successfully approaches towards making the Semantic Web a reality. The core idea of the Semantic Web vision [15], is to attach annotated information to the Web resources for describing instance data. This additional information should enable machines to process "the meaning of things" in a completely innovative way, to a be solution to *those problems and situations that we are yet to define* [1]. Such a novel infrastructure would enable

[1] `http://www.lassila.org/publications/2006/SCAI-2006-keynote.pdf`.

a new generation of Web applications far beyond the current technological constraints: for instance, machines will be able to automatically arrange travels collecting and coordinating schedule information for flights, trains, and booking hotels and excursions at the cost of a few clicks for the user. The enthusiasm around this research area is fully justified by the enormous potential power of this vision, which should increase the participation of machines to web activities: innovative web browsing automation/aid to complex searches [10], solving semantical optimization problems, improving information discover [53].

Although the first goal of the vision –wrapping semantic metadata around the content on the Web– is to a fair extent achieved, still, for the Semantic Web to show off its capabilities, some problems need to be better addressed. In particular, challenges come from the so called "Ontology Layer" which constitutes one of the main pillars of the Semantic Web scenario. Indeed, since this layer should allow to define shared and common domain theories, ontologies have gained special attention in that context, aiming at structuring underlying data for the purpose of semantic interoperability and comprehensive understanding between disparate sources and applications. More and more people and organizations have started to develop ontologies (see e.g. [3]); also, reasoners, applications and tools for ontologies (see e.g. [51, 95, 76]) are nowadays widespread.

To date, most research efforts have been spent on the analysis and development of sufficiently expressive languages and standards for the representation and querying of ontologies. However, querying efficiency has received attention only recently, especially for ontologies referring to large amounts of data. In this context, the reasoning mechanisms provided by the current RDF stores and SPARQL engines [2] come up with several major problems. This is especially true when querying the same data with respect to different ontologies and/or different entailment regimes are required. Indeed, the current implementations are not tailored for that kind of *dynamic* querying: most of them assume fixed datasets and usually reasoning is done *once and for all*, applying a materialization strategy at loading-time, that means that inference rules cannot be changed *on-the-spot*. As a consequence, current implementations do not properly behave in such a dynamically changing scenario, where inconsistency, ambiguities and wrong inferences on Web data inevitably pop up and thus must be taken into account. To date, these are open issues, still waiting for an efficient and effective resolution.

[2]i.e [1, 2, 5].

On the other hand, Answer Set Programming (ASP) [47] is a mature declarative modeling paradigm. It is based on sound theoretical foundations, versatile features and it provides interoperable and efficient solvers. In the Semantic Web view, both systems and languages based on ASP can be very interesting as suggested by recent initiatives aiming at placing ASP languages in the "Semantic Web Layer Cake" for addressing issues concerning ontological reasoning tasks delegated to the "Rule/Logic Layers" [38, 37]. Indeed, ASP has been proposed as the semantics of choice for defining a formalism for the "Rule Layer", with proper capabilities of interacting and reasoning on top of ontologies. Likewise, ASP can be exploited as a middleware formalism to which ontologies can be translated to, for solving reasoning tasks on translated ontologies (see e.g. [98]). It is then possible to take advantage of existing, advanced, computing methodologies for ASP: ASP systems allow to mix monotonic with non-monotonic reasoning, permit to combine rules with ontologies, and can interface external reasoners [39]. Notably, ASP is suitable for solving configuration and matchmaking problems involving reasoning with preferences by featuring easy to use, fully declarative soft and hard constraint specification languages [87]. Finally, ASP systems are scalable, and efficient implementation exist over DBMS that can act as a Semantic Web data repository for supporting semantic data storage, inferencing and querying, and thus becoming a viable platform for building semantic applications. Indeed, ASP extensions exist that are specifically tailored at Semantic Web applications, making the ASP paradigm ready for tackling many of the challenges the Semantic Web offers.

Objectives. The main purpose of this dissertation is to show how ASP can be fruitful used for implementing some of the building blocks of the Semantic Web , suggesting a way to answer to many of the pending questions the Semantic Web comprises.
In this perspective, we will focus on two different aspects, both necessary for dealing with Semantic Web applications, that are: *(i)* exploiting the ASP technology for building the necessary infrastructure for the Semantic Web applications as well as *(ii)* illustrate an ASP formalism which allows to model knowledge, and its semantics. As for the first point *(i)*, we will present a prototype triplestore system capable of persistently storing, querying and inferencing (with parametric inference semantics) over RDF datasets; as for the second point *(ii)*, we will show how ASP languages can be used to faithfully model and integrate knowledge expressed in RDFS and beyond, taking in consideration a formalization and implementation of Frame Logic [62] in a novel answer set semantics scenario.

Thesis overview and contributions

The work is split into three distinct pieces. The first part sets the preliminaries for subsequent discussions. In particular, we present the ASP approach [45] to logic programming, which differs from traditional logic programming. Indeed, it has a pure declarative semantics, (i.e. in contrast to Prolog programs [26]) and it is based on the notion of models as solutions for problems encoded in ASP languages. One more distinguished feature is that ASP allows for both strong and weak negation (also known as *negation as failure*). Due to negation as failure, the semantics for Answer Set programs are non-monotonic, that means that the set of logical consequences might decrease with increasing information in the program. In particular, we present an extension of the core language [99] which allows for *external* and *higher-order* atoms. Higher-order features are widely acknowledged as useful for performing meta-reasoning, among other tasks. Furthermore, the possibility to exchange knowledge with external sources in a fully declarative framework such as ASP is particularly important in view of applications in the Semantic Web area.

Afterwards, we introduce the most important aspects of the Semantic Web [15] platform that is oriented at adding a machine-readable meaning to Web pages and resources. This is achieved by using ontologies for a precise definition of shared terms and applying Knowledge Representation technology for automated reasoning tasks. We focus in particular on the middle layers (embracing the RDFS, Ontology/Rule Layers) dealing with expressing meta-data about Web resources, as well as defining terminological knowledge and assign semantics to concepts.

The main contribution of this thesis is presented in the second part of the dissertation where we address the problem of coupling the ASP methodology with Semantic Web technologies. Indeed, with respect to the Semantic Web, ASP can play the role of powerful rule language for facilitating sophisticated reasoning task as well as complementing the ontology formalisms. The contribution of our work in this setting consists in a thorough formalization of an ASP framework for dealing with Semantic Web data. In this sense, we formally introduce a translation of the RDFS semantic to ASP, that allows us for extending our approach to model the RDF query languages in terms of ASP. Once having defined such a suitable machinery for querying/reasoning on Semantic Web data, we focus on weaknesses and leaks that the current state-of-the-art systems reveal when dealing with a dynamically changing scenario. To this end, we study possible extensions of the query model for Semantic Web data. These latter are aimed at enabling a dynamical activation of inference rules and ontological schemas "on demand". Such formal results are practically experimented in a prototypical system (called GiaBATA) which is able to compete with the current systems, despite the overhead introduced by our proposed extensions.

The last part of the work focuses on ASP as a general methodology for modeling semantics of ontology languages. As a special case, we focus on Frame Logic [62, 107] introducing a novel framework for coping with frame-like syntax and higher-order reasoning within an Answer Set Programming environment. Our approach aims at closing the gap between Frame Logic based languages and Answer Set Programming , in both directions: on one hand, Answer Set Programming misses the useful Frame 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 Frame 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. Also, we show how it is possible to integrate multiple knowledge sources, with multiple semantics in the FAS framework.
The main contributions, briefly summarized, are as follows:

- successfully mixing the Answer Set Programming (ASP) formalism with semantic web technologies as basis for Semantic Web applications;
- efficiently dealing with mass-storage RDF data supporting dynamic inference and query answering;
- accomplishing challenging reasoning tasks in the Semantic Web , semantically modeling and integrating knowledge defined by expressive ontology languages.

We expect to provide relevant answers to the research community concerning the definition of new reasoning and query answering techniques (based on expressive query and rules languages), and the investigation of the interplay between ontology and rules languages.

Plan of the work

The three parts of the work are organized as follows. The first part provides some background notions which will be useful throughout the rest of the thesis: chapter 1 presents the Answer Set Programming formalism; chapter 2 introduces the main ideas behind the vision of the Semantic Web and its architecture. The central theme of the thesis is discussed in the third part. In particular, chapter 3 concentrates on possible usage of Answer Set Programming for Semantic Web applications: it formally presents a mapping of the problem of RDFS graph entailment to a corresponding problem of entailment under Answer Set Semantics, and extends that approach at translating the semantics of RDF query languages. Chapter 4 introduces a formal framework based on Answer Set Programming that allows for dynamically choosing the entailment regimes as well as ontology schema to be taken into account on a *per query* basis. Chapter 5 presents a fully ASP-based prototype system for managing RDFS and possibly higher semantics and practically deal with persistently stored data. The third part of the dissertation (chapter 6) investigates Answer Set Programming as a suitable basis for semantics modeling of expressive ontology languages, focussing on Frame Logic . Finally, we discuss related work and interesting directions for further developments, and draw the conclusions.

Preliminaries

This first part provides preliminary concepts which will be useful throughout this dissertation. The reader is introduced to background knowledge about two different topics in the field of Knowledge Representation and Reasoning, that we aim at integrating.
First, in chapter 1, we present the Answer Set Programming paradigm (ASP) that is a very successful descendant of a long tradition of logic programming formalisms. Thus, in the perspective of its fruitful usage for Semantic Web applications, we will focus on some Answer Set Programming advanced extensions. In particular, we will present extensions by *external* and *higher-order* atoms, conceived for dealing with external knowledge and meta reasoning, respectively.
Second, in chapter 2 we introduce the main ideas and fundamental pillars of the Semantic Web, which, at the moment of writing, is gaining momentum as the potential revolutionary technology.
In particular, this preliminary notions will be useful in the following of the dissertation aiming at showing how these two lines of research can benefit from each other.

Chapter 1

Answer Set Programming

This chapter recall basic notions on the ASP (ASP) paradigm and its semantics based on the notion of stable model, which is also the preferred semantics for Disjunctive Logic Programming (DLP). Indeed, DLP under the stable model semantics is a form of ASP widely appreciated as declarative modeling paradigm: it is based on sound theoretical foundations, features versatile, interoperable and efficient solvers, and has been applied successfully in a variety of contexts.
The basic idea is that a given problem is solved by devising a logic program such that the stable models of the program correspond to the solutions of the problem, which are then found by computing stable models for the program.
The success of ASP is much due to efficient solvers, such as DLV and SMODELS, which have been developed in recent years, coupling a formal treatment with an effective implementation for reasoning problems arising in a number of interesting application fields (i.e Knowledge Representation and Reasoning, Data Mining, Planning, Model Checking, and recently, Semantic Web).

1.1 Declarative Logic Programming

Logic programming in the narrowest sense can be traced back to debates in the late 1960s and early 1970s about declarative versus procedural representations of knowledge in Artificial Intelligence. Roughly speaking, programming languages belonging to the first paradigm, focus on how an algorithm solves a given problem, while in the purely declarative case, the programmer is only responsible for ensuring the truth of programs expressed in logical form, focussing on the problem, rather then on encoding the sequence of operations. Prolog [26], among others (Haskell, Lisp), is one of the most widespread tool for programming in logic. It is implemented as a sequential programming language, processing goals from left to right and selecting rules in textual order. Furthermore, Prolog provides extra-logical features to control the execution of the program. This means that the rule order as well as the predicate order within a rule can influence the program's result.

The paradigm that will be presented next, ASP [45], is somewhat different: it allows to state purely declarative logic programs, being based on view of program statements as constraints on the solution of a given problem. Subsequently, each model of the program encodes a solution to the program itself. More specifically, problems are represented in terms of (finite) theories, such that, the models of the latter determine the solutions of the original problem.

1.2 Logic Programs under Answer Set Semantics

ASP is a form of declarative programming paradigm. It stems from the stable model semantics of normal logic programs ([45]) line of research dealing with *negation as failure*. The latter is an extension to classical negation, denoting a fact as false if all attempts to prove it fail. Back in the 1990s, Gelfond and Lifschitz ([46]) proposed a logic programming approach that allows for both negation as failure as well as strong (or classical negation), subsequently extending ([47]) their semantics to *disjunction* in rule heads. As such, the answer set semantics extends the stable model semantics being defined on a syntactically richer class of programs. Indeed, the answer set semantics is defined for programs in which not only negation as failure may occur in program rules, but also strong negation and disjunctions. For an overview on other semantics for such programs see also [34].

According to the answer set semantics, a disjunctive logic program may have several alternative models (but possibly none), called *answer sets*, each corresponding to a possible view of the world. Indeed, one distinguished feature of the answer set semantics is its ability to generate multiple minimal models for a single problem specification.

ASP shows very interesting features which make it an attractive candidate for reasoning with knowledge. As mentioned, the paradigm is fully declarative and it allows for monotonic and non-monotonic reasoning, thus enabling default reasoning and non-monotonic inheritance. Moreover, the inherent nondeterminism make possible to define concepts ranging over a space of choices without any particular restriction. Therefore, it is a suitable formalism for handling incomplete and inconsistent information. Moreover, extension of the basic semantics with preferences, soft and hard constraint, enable the compact specification of search and optimization problems. Last but not least, ASP programs are decidable, that is, in their basic flavor, are naturally decidable: no special restrictions are needed in order to keep this important property. Disjunctive logic programs under answer sets semantics are very expressive. It was shown in ([36, 48]] that, under this semantics, disjunctive logic programs capture the complexity class $\Sigma_2^P(NP^{NP})$ (i.e., they allow us to express, in a precise mathematical sense, every property of finite structures over a function-free first-order structure that is decidable in nondeterministic polynomial time with an oracle in NP).

Interestingly, despite the computational expressiveness of ASP, state-of-the-art solvers currently reached the maturity for dealing with large datasets. Indeed, one of the main reasons for the increasing popularity of both the answer set semantics as well as the stable model semantics is in large part due to the availability of sophisticated solvers for these languages. Two prominent systems for computing answer sets are DLV ([40, 35, 64]) and SMODELS ([75, 93]), which allow for an efficient declarative problem solving. The DLV system, which is indirectly used in this thesis through the DLVHEX ([100, 99, 101]) and the DLV^{DB} framework ([103]), has been developed for over a decade as joint work of the University of Calabria and Vienna University of Technology and is still actively maintained.

1.2.1 Syntax of Answer Set Programs

Let $\mathcal{P}$, $\mathcal{C}$ and $\mathcal{X}$ be disjoint sets of predicate, constant, and variable symbols from a first-order vocabulary Φ, where $\mathcal{X}$ is infinite and $\mathcal{P}$ and $\mathcal{C}$ are countable. In accordance with Prolog's convention and common ASP solvers, we assume that elements from $\mathcal{C}$ and $\mathcal{P}$ are string constants that begin with a lowercase letter or are double-quoted, where elements from $\mathcal{C}$ can also be integer numbers. Elements from $\mathcal{X}$ begin with an uppercase letter and denote variables.

A *term* is either a variable or a constant. Given a predicate $p \in P$, an *atom* is defined as $p(t_1, \ldots, t_k)$, where k is called the arity of p and each $t_1, \ldots, t_k$ are terms. Atoms of arity $k = 0$ are called *propositional atoms*. A *classical literal* (or simply *literal*) l is an atom p or a negated atom $\neg p$, where $\neg$ is the symbol for true (or classical) negation. Its complementary literal is $\neg p$ (resp., p) (strongly negated literal or, simply, negated literal). A *negation* *as failure* literal (or *not -literal*) is a literal l or a default-negated literal $not\, l$. Negation as failure is a non-monotonic inference rule, used to derive $not\, L$ (i.e. that L is assumed not to hold) from failure to derive L. Thus, $not\, L$ evaluates to true if it cannot be demonstrated that L is true, i.e., if either L is false or we do not know whether L is true or false.

A *disjunctive rule* (*rule*, for short) r is a formula

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m. \qquad (1.1)$$

where $a_1 \vee \cdots \vee a_n$, $b_1, \ldots, b_k$ are classical literals. We say that $a_1 \vee \cdots \vee a_n$ is the *head* of r, while the conjunction $b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m$ is the *body* of r, where $b_1, \ldots, b_k$ (resp., $not\, b_{k+1}, \ldots, not\, b_m$) is the positive (resp., negative) body of r. We use $H(r)$ to denote its head literals, and $B(r)$ to denote the set of all its body literals $B(r)^+ \cup B(r)^-$, where $B(r)^+ = \{b_1, \ldots, b_k\}$ and $B(r)^- = \{b_{k+1}, \ldots, b_m\}$. A rule with empty body is called *fact* and we often omit "$\leftarrow$", while a rule with empty head is an *integrity constraint*. A rule with exactly one head literal is a *normal rule*.

A *disjunctive logic program* (or simply *program*) P is a finite set of rules r of the form 1.1. A not-free program P (i.e., such that $\forall r \in P : B^-(r) = \emptyset$) is called *positive*, and a $\vee$-free program P (i.e., such that $\forall r \in P : \mid H(r) \mid \leq 1$) is called Datalog program (or normal logic program).

1.2.2 Semantics of Answer Set Programs

The semantics of disjunctive logic programs is defined for variable-free programs. Thus, we first define the *ground instantiation* of a program that eliminates its variables.

The *Herbrand universe* of a program P , denoted HU_P , is the set of all constant symbols $C \subset \mathcal{C}$ appearing in P . If there is no such constant symbol, then $HU_P = \{c\}$, where c is an arbitrary constant symbol from Φ. As usual, terms, atoms, literals, rules, programs, etc. are *ground* iff they do not contain any variables. The *Herbrand base* of a program P, denoted HB_P, is the set of all ground (classical) literals that can be constructed from the predicate symbols appearing in P and the constant symbols in HU_P . A *ground instance* of a rule $r \in P$ is obtained from r by replacing every variable that occurs in r by a constant symbol from HU_P . We use $grnd(P)$ to denote the set of all ground instances of rules in P .

The semantics for disjunctive logic programs is defined first for positive ground programs. A set of literals $X \subseteq HB_P$ is *consistent* iff $\{p, \neg p\} \not\subseteq X$ for every atom $p \in HB_P$. An *interpretation I relative to* a program P is a consistent subset of HB_P . We say that a set of literals S *satisfies* a rule

r if $H(r) \cap S \neq \emptyset$ whenever $B^+(r) \subseteq S$ and $B^-(r) \cap S = \emptyset$. A *model* of a positive program P is an interpretation $I \subseteq HB_P$ such that I satisfies all rules in P. An *answer set* of a positive program P is the least model of P w.r.t. set inclusion.
To extend this definition to programs with negation as failure, we define the *Gelfond-Lifschitz transform* (also called the *Gelfond-Lifschitz reduct*) of a program P relative to an interpretation $I \subseteq HB_P$, denoted P^I, as the ground positive program that is obtained from $grnd(P)$ by

- deleting every rule r such that $B^-(r) \cap I = \emptyset$, and
- deleting the negative body from every remaining rule.

An *answer set* of a program P is an interpretation $I \subseteq HB_P$ such that I is an answer set of P^I .

Example 1.2.1 *Consider the following program P:*

$$p \leftarrow not\, q.$$
$$q \leftarrow not\, p.$$

Let $I_1 = \{p\}$; then $P^{I_1} = \{p\}$ with the unique model $\{p\}$ and thus I_1 is an answer set of P. Likewise, P has an answer set $\{q\}$. However, the empty set $\emptyset$ is not an answer set of P, since the respective reduct would be $\{p; q\}$ with the model $\{p; q\}$.
□

1.3 ASP^{HEX}: an extension of ASP

In this section we focus on ASP^{HEX} programs, which are non-monotonic logic programs under the answer set semantics admitting *external* and *higher-order* atoms.
Intuitively, an *higher-order* atom allows to quantify values over predicate names, and to freely exchange predicate symbols with constant symbols, like in the rule:

$$C(X) \leftarrow subClassOf(D, C), D(X). \tag{1.2}$$

An *external* atom facilitates to determine the truth value of an atom through an external source of computation. Informally, an external atom models knowledge that is external to a given logic program, and whose extension might be infinite. This feature is realized by the introduction of "parametric" external predicates, whose extension is not specified by means of a logic program but implicitly computed through external code. For instance, the rule:

$$reached(X) \leftarrow \&reach[edge, a](X). \tag{1.3}$$

computes the predicate $reached$ taking values from the predicate $\&reach$, which computes 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, etc.).
In our setting, the usage of external atoms is crucial for modeling some aspects of the normative language for querying semantic web data (see chapter 3) that require to deal with infinite sets and/or that are difficult to be encoded using plain ASP. The higher-order capabilities are necessary to enabling meta reasoning for Semantic Web applications or for meta-interpretation in ASP itself, etc.

1.3.1 Syntax of ASP^{HEX} Programs

Let $\mathcal{C}$, $\mathcal{X}$ and $\mathcal{G}$ mutually disjoint sets whose elements are called constant, variable and external predicate names, respectively. $\mathcal{C}$ may be infinite. Unless explicitly specified, elements from $\mathcal{X}$ (resp. $\mathcal{C}$) are denoted with first letter in upper case (resp., lower case), while elements from $\mathcal{G}$ are prefixed with "&". We note that constant names serve both as individual and predicate names. Elements from $\mathcal{C} \cup \mathcal{X}$ are called *terms*. An *higher-order atom* (or *atom*) is a tuple $(Y_0, Y_1, \cdots, Y_n)$, where $(Y_0, Y_1, \cdots, Y_n)$ are terms; $n \geq 0$ is the arity of the atom. Intuitively, Y_0 is the predicate name, and we thus also use the more familiar notation $Y_0(Y_1, \cdots, Y_n)$. The atom is *ordinary*, if Y_0 is a constant. For example, $(x, rdf : type, c)$, $node(X)$, and $D(a, b)$, are atoms; the first two are ordinary atoms.

An *external atom* is of the form

$$\&g[Y_1, \cdots, Y_n](X_1, \cdots, X_m), \tag{1.4}$$

where $(Y_1, \cdots, Y_n)$ and $(X_1, \cdots, X_m)$ are two lists of terms (called *input* and *output* lists, respectively) , and $\&g \in \mathcal{G}$ 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. 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 *rule* r is of the form:

$$a_1 \vee \cdots \vee a_n \leftarrow b_1, \ldots, b_k, not\, b_{k+1}, \ldots, not\, b_m. \tag{1.5}$$

where $m, k \geq 0$, $a_1 \vee \cdots \vee a_n$ are atoms, and $b_1, \ldots, b_k$ are either atoms or external atoms. We define $H(r) = \{a_1 \vee \cdots \vee a_n\}$ and $B(r) = B^+(r) \cup B^-(r)$, where $B(r)^+ = \{b_1, \ldots, b_k\}$ and $B^-(r) = \{b_{k+1}, \ldots, b_m\}$. If $H(r) = \emptyset$ and $B(r) \neq \emptyset$, then r is a *constraint*; while if $B(r) = \emptyset$ and $H(r) \neq \emptyset$, then r is a *fact*; r is *ordinary* if it contains only ordinary atoms.

An ASP^{HEX} program is a finite set P of rules. It is *ordinary*, if all rules are ordinary.

1.3.2 Semantics of ASP^{HEX} Programs

The semantics of ASP^{HEX} programs generalizes the answer set semantics ([47]) by extending it to external atoms. In particular, here we use the notion of a reduct as defined by Faber et al. (referred to as *FLP-reduct* henceforth [41]) instead of to the traditional reduct by Gelfond and Lifschitz ([47]).
Let P be an ASP^{HEX} program. The *Herbrand base* of P, denoted HB_P, is the set of all possible ground versions of atoms and external atoms occurring in P obtained by replacing variables with constants in $\mathcal{C}$. The *grounding* of a rule r, $grnd(r)$, is defined accordingly, and the groundingly of a program$P$ is given by $grnd(P) = \bigcup_{r \in P} grnd(r)$. Unless specified otherwise, $\mathcal{C}$, $\mathcal{X}$ and $\mathcal{G}$ are implicitly given by P.

Example 1.3.1 *[39] Given $\mathcal{C} = \{edge, arc, a, b\}$, ground instances of $E(X, b)$ are for instance $edge(a, b), arc(a, b), a(edge, b), arc(arc, b)$; ground instances of $\&reach[edge, N](X)$ are all possible combinations where N and $X \in \mathcal{C}$, for instance $\&reach[edge, edge](a), \&reach[edge, arc](b), \&reach[edge, edge](edge)$, etc.*

An *interpretation relative to P* is any subset $I \subseteq HB_P$ containing only atoms. We say that I is a *model* of atom $a \in HB_P$, denoted $I \models a$, if $a \in I$. With every external predicate name $\&g \in \mathcal{G}$, we associate an $(n+m+1)$-ary boolean function $f_{\&g}$ assigning each tuple $(I, y_1, \ldots, y_n, x_1, \ldots, x_m)$ either 0 or 1, where $n = in(\&g)$, $m = out(\&g)$, $I \subseteq HB_P$, and $x_i, y_j \in \mathcal{C}$. We say that $I \subseteq HB_P$ is a *model* of a ground external atom $a = \&g[y_1, \ldots, y_n](x_1, \ldots, x_m)$, denoted $I \models a$, iff $f_{\&g}(I, y_1, \ldots, y_n, x_1, \ldots, x_m) = 1$.

Example 1.3.2 *[39] Let us associate with the external atom* $\&reach$ *a function* $f_{\&reach}(I, E, A, B) = 1$ *iff* B *is reachable in the graph* E *from* A*. Let* $I = \{e(b, c), e(c, d)\}$*. Then,* I *is a model of* $\&reach[e, b](d)$ *since* $f_{\&reach}(I, e, b, d) = 1$.

Let r be a ground rule. We define (i) $I \models H(r)$ iff there is some $a \in H(r)$ such that $I \models a$, (ii) $I \models B(r)$ iff $I \models a$ for all $a \in B^+(r)$ and $I \not\models a$ for all $a \in B^-(r)$, and (iii) $I \models r$ if $I \models H(r)$ whenever $I \models B(r)$. We say that I is a *model* of an ASP^{HEX} program P , denoted $I \models P$, iff $I \models r$ for all $r \in grnd(P)$. We call P *satisfiable*, if it has some model.

Given a ASP^{HEX} program P , the *FLP-reduct* of P with respect to $I \subseteq HB_P$, denoted fP^I , is the set of all $r \in grnd(P)$ such that $I \models B(r)$. $I \subseteq HB_P$ is an answer set of P iff I is a minimal model of fP^I . For more details, cf. [39]. It is worth noting that, external predicates may be associated with functions having an unknown (and possibly infinite) co-domain. These are computed by means of an associated evaluation function (*oracle*). Unfortunately, in a context where value invention is explicitly allowed, grounding a program against an infinite set of symbols leads to an infinite ground program, which obviously cannot be built in practice. Thus $grnd(P)$ is in principle an infinite program. However a large class of ASP^{HEX} programs is proven to have finite answer sets, namely *vi-restricted* (*value invention-restricted*) programs, whose definition is given in [24].

Chapter 2

The Semantic Web

In this chapter we introduce the basic notions about the Semantic Web by providing motivation, description, and application examples. Among all, we focus on the standard, languages and technologies conceived for the RDF(S)/Ontology Layers of its architecture.
In the perspective of envisaging further evolvement of the vision behind the Semantic Web, we discuss possible usage of rule-based formalisms as a complementary tool to be used in ontology languages, either in conjunction with or as an alternative to expressive logics, other than for drawing inferences, expressing constraints, etc. This will provides justifications to our setting (presented in chapter 3) which relies on the Answer Set Programming as the rule-based formalism to cope with several open issues in the context of the Semantic Web .

2.1 Historical background

The European Organization for Nuclear Research in Switzerland, CERN, is where it all began in March 1989. A physicist, Tim Berners-Lee, wrote a proposal about *a large hypertext database with typed links*, showing how information could be easily transferred over the Internet by using *hypertext*, the very well known "point and click" system for navigation through information. In 1990 Tim Berners-Lee and Robert Cailliau presented that idea to the European Conference on Hypertext Technology. By Christmas of the same year, all the tools necessary for a working Web were ready. But the real debut-day of the Web as a publicly accessible service on the Internet, has been market at August 6 1991, when Berners-Lee posted a short summary of the *World Wide Web* (WWW) project on the `alt.hypertext newsgroup.`

" The World Wide Web (WWW) project aims to allow all links to be made to any information anywhere. [...] The WWW project was started to allow high energy physicists to share data, news, and documentation. We are very interested in spreading the web to other areas, and having gateway servers for other data. Collaborators welcome! "
Therefore, the idea behind the project was to connect hypertext with the Internet and personal computers, realizing a single information network to help CERN physicists in sharing all the information, with the possibility to browse between web pages using links. Early adopters of the World Wide Web have been primarily university-based scientific departments or physics laboratories.
Afterward, in 1994, the *World Wide Web Consortium* (W3C) [1] started its standardization work to improve the quality of the Web. Fundamental standards for the Web have been settled, such as *Hypertext Transfer Protocol* (HTTP) for transferring arbitrary data over networks, *Uniform Resource Identifier* (URI) to link Web resources, and *Hypertext Markup Language (HTML)* for representing hypertext with URIs. Suddenly, new requirements come into the light, posing questions especially about the data formats to be used for exchanging data on the web for the purpose on syntactic and semantics interoperability.

The first turning point, in order to deal with the amount of mostly unstructured information available on the web, goes back to 1998, when the W3C presented the first standard for marking up data, the so called *Extensible Markup Language* (XML) [2]. Opposed to HTML, which is a markup-language for a specific kind of hypertext documents, XML has been intended as a markup-language for arbitrary document structure, aiming at separation of content from presentation and being flexible enough to support platform and architecture independent data interchange. Indeed, the semi-structured data-model [7] provided via XML allows to create a vocabulary, and use this vocabulary to describe data itself. However, when it comes to semantic interoperability, XML reveals several disadvantages, since it aims at the structure of documents and does not impose any common interpretation of the data contained in the document.

A year later, a new specification, the *Resource Description Framework* (RDF) [3] ([42]), was designed at W3C with the intend to provide a general formalism for conceptual description and for automatic data processing of meta-information. In the 2001, the book [14] and the article [15] make

[1]http://www.w3.org/.
[2]http://www.w3.org/XML/.
[3]http://www.w3.org/RDF/.

one more step towards semantic interoperability introducing the idea the *Semantic Web* .

" Adding semantics to the web involves two things: allowing documents which have information in machine-readable forms, and allowing links to be created with relationship values. Only when we have this extra level of semantics will we be able to use computer power to help us exploit the information to a greater extent than our own reading. " According to Tim Berners-Lee the Web with all its content was mature and well-established among Internet users, but what was missing was a *Web of Data*, allowing for machine-readable information and automatic data processing. Since 2001 the *W3C Semantic Web Activity* [4] is actively working on this "web of data".
Summarizing, the vision of the Semantic Web is to extend principles of the Web from documents to data, aiming at creating a common framework that allows machines for supporting automatic processing of data, as well as sharing and reusing them across applications. In this sense, whereas the original Web is interested in (interchange of) documents, the attention is now moved to the data, making of utter importance the capability to define and describe the relations among resources on the Web.

2.2 The Semantic Web in practise

Like other technologies, the interest in creating and developing the Semantic Web is motivated by the opportunities it might bring: either it can solve new problems, or it can solve old problem in a better way. One very promising application area of Semantic Web technology is the field of Knowledge Management. Indeed, the Knowledge Management traditionally identified as key in maintaining the competitiveness of organizations, is now facing new problems triggered by the web: information overload, inefficient keyword searching, heterogeneous information integration and so on. These problems are being tackled by the Semantic Web technology, which focuses on acquiring, maintaining and accessing structured information source. The possibility to properly use ontologies to describe web resources, and to represent knowledge on the Web in a structured, logical, and semantic way, can change the way that agents can navigate, harvest and utilize information on the Web ([78]). In this new perspective, agents could be allowed to migrate from one site to another, carrying their codes, data, running states, and intelligence to fulfill their missions

[4] http://www.w3.org/2001/sw/.

autonomously and intelligently. These new technologies can be applied for automatic web-service discovery, invocation, composition and interoperation as well, thus enhancing the landscape of electronic commerce [55]. Other examples belong to the area of Social networking ([22, 16, 97], Personal information management [5], Information syndication ([88, 52]), Library/museum data [3], Network security and configuration [6].

Moving from theory to practice there are already a number of active projects, usually aimed at some specific domain of interest. Indeed, after being picked up by the Open Source community the Semantic Web has been used by small and specialized startups and finally by business in general. There are a number of projects in the running, among them editors, content management systems, reasoners, etc [7].
For the sake of this work we are interested in the systems commonly called RDF "triplestore" ([2, 5, 1]) [8], offering capabilities of RDF(S) storage and querying thus playing the role of database. However, an important difference with respect to traditional relational databases, is that a triplestore might represent information not explicitly stored, and which can be obtained by logical inference. Allowed logical inference is usually specified in terms of entailment rules, that might be normative (coming from the RDF(S) semantics specifications), but also a subset of normative ones (such as the so-called ρDF [74] fragment of RDF(S)) or user defined ones.

2.3 Basic Building Blocks

The development of the Semantic Web proceeds in layers of Web technologies and standards, where every layer is built on top of lower layers, namely: *URI +Unicode*, *XML*, *RDF + RDF Schema*, *OWL*, *Logic*, *Proof*, *Trust*, and *Digital Signature* (see fig. 2.1). The key element of this design is the use of Web addresses (URIs) to identify Web resources and for representing text in a uniform way over different computer platforms and natural languages (Unicode). This allows for establishing relationships between any two resources. On the next level, the XML framework is used for annotating information to assist data sharing. In order to provide a precise definition of shared terms in Web resources, it is needed to define stan-

[5]i.e OSAF:Chandler, at `www.osafoundation.org/`.

[6]i.e SWAD-Europe, at `http://www.w3.org/2001/sw/Europe/`.

[7]cf. http://esw.w3.org/topic/SemanticWebTools#preview for a comprehensive and up-to-date list.

[8]cf. http://esw.w3.org/topic/SemanticWebTools#preview for RDF TripleStores and http://esw.w3.org/topic/SparqlImplementations for SPARQL Implementations.

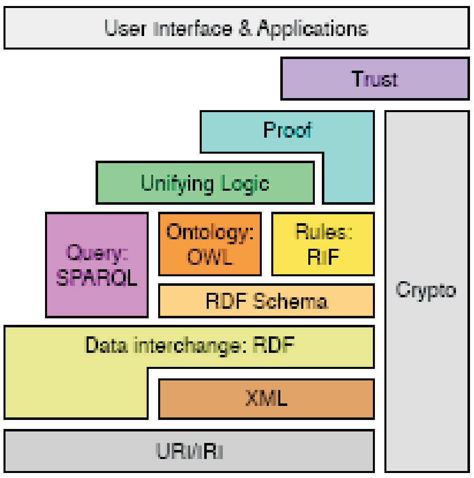

Figure 2.1: The Semantic Web Architecture.

dards and languages describing the structure of the knowledge published. Therefore, the next step after making the data on the Web machine processable, is facilitating the direct interaction of applications over the Web, using Knowledge Representation technology for automated reasoning on Web resources and applying cooperative agent technology for processing information and facilitating knowledge sharing and exchanging. There is currently a lot of research work on these aspects, mainly involving the three consecutive RDF(S)-Ontology-Rule layers (listed from bottom to top) that we focus on.

2.3.1 The RDF(S) Layer

The *Resource Description Framework* (RDF) [42], is the basic language for expressing data models, which refers to resources and their relationships. As such, it represents the standard for data interchange on the Web. RDF is designed to represent information in a minimally constraining and flexible way. RDF uses URIs to name the resources and the relationship between them, allowing anyone to make statements about any resource. Such a kind of statement of a relationship between resources encodes a so called "triple", consisting of a *subject*, a *predicate* (also called a *property*) and an *object*. Intuitively, an assertion of an RDF triple says that some relationship, indicated by the predicate, holds between the things denoted by subject and object of the triple. This linking structure reflects a simple "graph-based data model", that forms a directed, labeled graph, where the edges represent the named link between two resources, represented by the graph

nodes.
RDF has a formal semantics which provides a basis for reasoning about the meaning of an RDF expression. In particular, it supports rigorously defined notions of entailment which provides a basis for defining reliable rules of inference in RDF data. Informally, the assertion of an RDF graph amounts to asserting all the triples in it, so the meaning of an RDF graph is the conjunction (logical AND) of the statements corresponding to all the triples it contains. A formal account of the meaning of RDF graphs is given in [80].

RDF Schemas (shortly RDFS) [27] is used to declare vocabularies, that means, the sets of semantics property-types defined by a particular community. Therefore, RDFS it alloes for defining the valid properties in a given RDF description, as well as any characteristics or restrictions of the property-type values themselves. Indeed, as semantic extension of RDF, RDFS defines classes and properties that may be used to describe classes, properties and other resources, with semantics for generalized hierarchies of such properties and classes. Thus, RDFS figures as an extensible knowledge representation language, providing basic elements for the description of RDF ontologies. The semantics of RDFS is defined through a set of axiomatic triples and entailment rules that determine the full set of valid inferences from an RDF graph.
Summarizing, what RDF (and on top) languages achieve is placing meaning directly within the data, rather than within the code of the program which processes the data. Moreover, relying on web-based URI references and adopting the mentioned graph-based model, the resulting data is connected into a vast network of 'meaning' data across the internet. That allows data to be structured in networks of nodes which can be easily merged, giving the Semantic Web the potential to evolve into a global online web of data.

2.3.2 The Ontology Layer

On top of the RDF(S) layer is built the Ontology Layer which embraces ontology vocabularies aiming at expressing terminological knowledge and providing semantics to concepts. In particular, the W3C has chosen a language, the *Ontology Web Language* (OWL) [77, 69], which is close to a syntactic variant of an expressive but still decidable Description Logic (*DL*) [11], namely $\mathcal{SHOIN}(\mathcal{D})$. OWL is divided in three increasingly expressive languages called OWL Lite, OWL DL, and OWL Full, where OWL DL and OWL Lite are based upon the *DLs* $\mathcal{SHOIN}(\mathcal{D})$ and $\mathcal{SHIF}(\mathcal{D})$, resp.

The syntactic form of an OWL knowledge base is compliant with the RDF format, where some of the keywords of the language are enriched with additional meaning. In particular, OWL allows to specify transitive, symmetric, functional, inverse, and inverse functional properties. Moreover, OWL admits specifications of complex class descriptions that provided in terms of union or intersection of other classes, as well as restrictions on properties. Finally, OWL supports the explicit definition of equality or inequality relations between individuals.
A new revision of the OWL DL Ontology Language, called OWL 1.1 has been proposed in [12]. In October 2009, an extended version (namely OWL 2 [70]) with several new features as been announced at W3C. Motivated by application requirements, OWL 2 extends the OWL language with a small but useful set of features, including extra syntactic sugar, additional property and qualified cardinality constructors, extended datatype support, simple meta-modeling, and extended annotations.

Remark. The payload of the expressiveness of OWL is, unfortunately, the high computational cost associated to many of the reasoning tasks commonly performed over an ontology. Nonetheless, a variety of Web applications require highly scalable processing of data, more than expressiveness. This puts the focus back to the lower RDF(S) data layer. In this context, RDF(S) should play the role of a lightweight ontology language. In fact, RDF(S) has few and simple descriptive capabilities (mainly, the possibility to describe and reason over monotonic taxonomies of objects and properties). One can thus expect from RDF(S) query systems the ability of querying very large datasets with excellent performance, yet allowing limited reasoning capabilities on the same data. In fact, as soon as the RDF(S) format for data has been settled, research focussed on how RDF(S) can be fruitfully stored, exchanged and queried.

2.3.3 The SPARQL Protocol Query Language

The W3C *Data Access Working Group* (DAWG) has developed the SPARQL query language [9]. The name is a recursive acronym that stands for SPARQL Protocol and RDF Query Language. As the name implies, SPARQL is a general term for both a protocol and a query language. Indeed, SPARQL can be used as part of a general programming environment, but queries can also be sent as messages to a remote SPARQL endpoints using the

[9] http://www.w3.org/TR/rdf-sparql-query/, W3C Recommendation.

companion technologies SPARQL Protocol (a method for remote invocation of SPARQL queries) and SPARQL Query Result in XML. Here we focus on the SPARQL query language, whose syntax and semantics is defined in cf. [85, 79, 9].

SPARQL is a syntactically-SQL-like language for querying RDF graphs. Similar in spirit to SQL, which allows to extract, combine and filter data from relational database tables, SPARQL allows to extract, combine and filter data from RDF graphs. In particular, SPARQL defines queries in terms of graph patterns that are matched against the directed graph representing the RDF data. The semantics and implementation of SPARQL involves, compared to SQL, several peculiarities, which we do not focus on, cf. [85, 79, 81, 84] for details. SPARQL has the same expressive power as non-recursive Datalog ([81, 9]) and includes a set of built-in predicates in so called **`filter`** expressions. SPARQL allows for querying required and **`optional`** graph patterns along with their conjunctions and disjunctions. The result of the match can also be used to construct new RDF graphs using separate graph patterns. Variables may occur in the predicate position to query unknown relationships, and the **`optional`** keyword provides support for querying relationships that may or may not occur in the data. Interestingly, the SPARQL **`graph`** keyword allows data to be queried along with its provenance information; that means that it can be used to discover the URI of the graph that contains the data that matches the query. This allows a single query to join information from multiple data sources accessible across different Web sites.
The SPARQL specification defines the results of queries based on RDF simple entailment. However, the overall SPARQL design can be used for queries which assume a more elaborate form of entailment, but this is still an open research problem (regimes other than simple entailment are left open in the SPARQL specification).

Answer Set Programming for the Semantic Web

Thanks to initiatives such as DBPedia or the Linked Open Data project,[10] a huge amount of machine-readable RDF [42] data is available, accompanying pervasive ontologies describing this data such as FOAF [22], [16], or YAGO [97]. A vast amount of Semantic Web data uses rather small and lightweight ontologies that can be dealt with rule-based RDFS and OWL reasoning [102, 74, 57], in contrast to the full power of expressive description logic reasoning. However, even if many practical use cases do not require complete reasoning on the terminological level provided by DL-reasoners, the following tasks become of utter importance.

First, RDFS as a lightweight ontology language is gaining popularity and, consequently, tools for scalable RDFS inference and querying are needed, capable to handle and evaluate queries on large amounts of RDF instance data.

Second, a Semantic Web system should be able to take into account implicit knowledge found by ontological inferences as well as by additional custom rules involving built-ins or even non-monotonicity. The latter features are necessary, e.g., for modeling complex mappings [83] between different RDF vocabularies.

As a third point, joining the first and the second task, if we want the Semantic Web to be true we need triplestores that allow dynamic querying of different data graphs, ontologies, and (mapping) rules harvested from the Web. Use cases for such dynamic querying involve, e.g., querying data with different versions of ontologies or queries over data expressed in related ontologies adding custom mappings (using rules or "bridging" ontologies).

In the following we will address the above mentioned questions, following the line of reasoning described next. First, in chapter 3, we show how RDF graphs, and the corresponding RDFS entailment rules can be faith-

[10] `http://dbpedia.org/` and `http://linkeddata.org/`.

ful translated to corresponding logic programs. We thus show how the problem of RDFS graph entailment can be mapped to a corresponding problem of entailment under answer set semantics. Note that RDF query languages, and notably, SPARQL [85], are defined in terms of graph entailment. Thus, this first results constitutes the important basis for translating the semantics of RDF query languages to corresponding ASP programs. In chapter 4 we first introduce SPARQL along with RDF(S) and partial OWL inference by means of some motivating example queries which existing systems partially cannot deal in a reasonably manner. Then, we show how the SPARQL language can be enhanced with custom ruleset specifications and arbitrary graph merging specifications. Finally, chapter 5 proposes a fully ASP-based framework, showing how it is possible managing the RDFS and possibly higher semantics and practically deal with persistently stored data.

Chapter 3

ASP for Semantic Web applications

3.1 Using ASP to deal with RDF(S)

Here we present the first attempt to a faithful translation of the whole normative RDF(S) into ASP. The problems arise from some important semantic differences between the two languages that briefly summarize.

First, particular attention must be devoted to the usage of *blank nodes*, that acting as anonymous variables might, apparently, require the adoption of a language with existential constructs. A blank node, indeed, can be seen as a limited form of existential quantification. ASP semantics is usually derived from function-free Datalog, and has no direct possibility to deal with unnamed individuals (objects whose existence is known but whose identity cannot be reconduced to a known constant symbol) in a context where unique name assumption is not assumed. Nevertheless, we show that blank nodes can be transformed into either anonymous constant symbols or universally quantified variables.

The second important issue concerns the problem of *inconsistency*. Most deductive databases, founded on stratified Datalog, do not allow modeling of contradiction as first order logic does. A stratified Datalog program has always a model (possibly empty), whereas RDF graphs might be inconsistent in a first order sense (i.e. following the *ex-falso quodlibet* principle). Conversely, this is not a problem under stable models semantics (rather than stratified Datalog), applied to non-stratified programs under brave or cautious reasoning. Indeed, constraints (which are, under stable model semantics, particular, non-stratified, programs) allow modeling inconsistency in a similar way as first order logic does.

Finally, treatment of normative *axiomatic triples* reveals several problems. Although modeling the set A of infinite triples might raise practical concerns, we show how these problems can be circumvented by restricting "relevant" axiomatic triples to a finite subset.
In the following, we provide the full translation of RDF(S) (especially of its entailment rules) into a suitable extension of ASP^{HEX} (ASP extended with external and higher-order predicates) that has been recently proposed.

3.1.1 The RDFS Semantics

The semantics of RDF(S) is outlined below, following the normative specification given in [80].

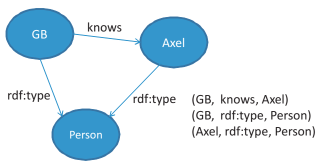

Figure 3.1: The example graph G_1' and its corresponding set of triples.

Preliminaries. Let I, B, and L denote pairwise disjoint infinite sets of IRIs (Internationalized Resource Identifiers), blank nodes, and RDF literals, respectively. An RDF *tripleset* or *RDF graph* G (or simply *graph*) is defined as a set of *triples* (s, p, o) from $I \cup B \times I \cup B \times I \cup B \cup L$ (cf. [79, 9]) [1]; s is called the *subject*, p the *predicate* and o the *object* of the triple, respectively. By $blank(G)$ we denote the set of blank nodes of G.[2] As commonly done in the literature (see, e.g. [74], we enlarge our focus to graphs where literals are allowed to appear also in the subject position within a triple. We occasionally denote a blank node $b \in B$ as starting with prefix "_", such as _*b*. The meaning of RDF(S) graphs is given in terms of first order interpretations without the unique name assumption. As an example, the triple (_*b*,*hasName*,*GB*), can be seen as the first order sentence $\exists B\ hasName(B, GB)$, that is, "there exists an object in the domain of discourse having name *GB*". The first order semantics of RDF(S) is difficult to be implemented in practice without a concrete proof theory. Thus, the official specification of RDF(S) states equivalence theorems between the first order semantics and the notion of *graph entailment*. In the following we directly define RDF(S) semantics in

[1]In practice, I, B and L are strings of a given vocabulary and are subject to syntactic rules.

[2]Note that we allow *generalized RDF graphs* that may have blank nodes in property position.

terms of graph entailment, referring to the two notions of *simple* entailment and *RDFS* entailment[3].

Intuitively, graph entailment is built on the notion of subgraph isomorphism, that is, it amounts to finding a function mapping a graph to another. This mapping function must have the following characteristics. A *mapping* μ is a function mapping elements from $I \cup B \cup L$ to elements in the same set, subject to the restriction that an element $i \in I \cup L$ must be such that $\mu(i) = i$. We indicate as $\mu(G)$ the set of triples $\{(\mu(s), \mu(p), \mu(o)) \mid (s, p, o) \in G\}$. Note that elements of B (blank nodes), are used to model existential quantification and can be mapped to any element, while elements of I and L preserve their identity through the mappings.

Simple and RDFS entailment. Given two graphs G_1 and G_2, we say that $G_1 \models G_2$ (G_1 *simply entails* G_2) if there is a mapping μ such that $\mu(G_2)$ is a subgraph of G_1. In general, deciding this kind of entailment is NP-complete [80] with respect to the combined sizes of G_1 and G_2, while it has been observed that entailment is polynomial in the size of G_1 [30].
For instance, the graph $G_2' = \{(\mathit{GB}, \mathit{knows}, \_b)\}$, where $\_b$ is a blank node, is entailed by the graph in Figure 3.1 (a matching mapping is obtained by associating $\_b$ to *Axel*). Notably, blank nodes in G_2' are seen as variables to be matched, not differently from variables in an SQL-like query language as actually SPARQL is.

Simple entailment is not directly used as a notion for defining semantics for RDF graphs. In fact, the two notions of RDF-entailment and RDFS-entailment, are originally defined by means of first order logic. However, the RDFS-entailment Lemma (sec. 7.3 of [80]) brings back RDFS-entailment of two graphs G_1 and G_2 to simple entailment between a graph $R(G_1)$ and G_2. The graph $R(G_1)$ is the *closure* of G_1, which is built *(i)* by applying, until saturation, so called RDF and RDFS entailment rules, and *(ii)* by adding normative axiomatic triples to G_1. RDF and RDFS entailment rules are reported in Figure 3.3 (the reader is referred to [80], sec. 7.3, for all details). Axiomatic triples are composed of a small finite set (which is tabulated in [80]) and an infinite portion, as we explicitly show in Figure 3.2.
We can classify normative RDFS semantics as follows:

• Taxonomy rules. Such rules mainly regard the keywords *rdfs:subClassOf* and *rdfs:subPropertyOf*. Two separate taxonomies can be defined in an RDFS graph: a taxonomy of classes (a class c is a set of individuals C

[3][80] includes also the notion of RDF-entailment and D-entailment.

such that *(i, rdf:type, c)* holds for each $i \in C$), and a taxonomy of properties (a property p is a binary relation P, where each couple $(s, o) \in P$ is encoded by the triple *(s,p,o)*). *rdfs:subClassOf* and *rdfs:subPropertyOf* are the special keywords to be used for defining such taxonomies. The semantic properties of these keywords are enforced by entailment rules such as RDFS5,7,9 and 11 (see Figure 3.3), which implement a simple (and monotonic) inheritance inference mechanism. For instance, if a graph G contains the triple *(person,rdfs:subClassOf,animal)* and the triple *(g,rdf:type,person)* then the closure $R(G)$ must contain *(g,rdf:type,animal)* as prescribed by entailment rule RDFS9.

• Typing rules. The second category of entailment rules strictly regard properties: the *rdfs:range* and *rdfs:domain* keywords allow the declaration of the class type of s and o for a given couple (s, o) when *(s,p,o)* holds (rules RDFS2 and 3 as well as IRDFS2 and 3), while rules RDF2A,2B, RDFS1A and 1B assess literal values[4].
For instance, from *(g,hasFather,c)* and *(hasFather,rdfs:domain,person)* one can infer *(c,rdf:type,person)*, by means of rule RDFS2. Note that the application of typing rules (and of the RDFS entailment rules in general) cannot lead to contradiction. This has two important consequences: first, range and domain specifications are not seen as integrity constraints, as it is usually assumed in the database field. Second, a triple graph cannot, usually, contain contradictory information. In fact, inconsistency is triggered only if a graph contains some ill-formed literal (or XMLLiteral) (e.g. a constant symbol $l \in L$ of type *rdfs:Literal* or *rdf:XMLLiteral*, which does not comply with syntactic prescriptions for this type of objects). In such a case, a graph G is assumed to be inconsistent (rules RDF2B and RDFS1B), and it is normatively prescribed that $R(G)$ must coincide with the set of all the possible triples.

• Axiomatic triples. These triples hold "a priori" in the closure of any graph, and give special behavior to some other keywords of the language. For instance, the triple *(rdfs:subClassOf, rdfs:domain, rdfs:Class)* enforces the domain of the property *rdfs:subClassOf* to be *rdfs:Class*. In particular, axiomatic triples contain an infinite subset (shown in Figure 3.2) which we call A. This set of triples regards special keywords used for denoting collections of objects (*Containers*) such as Bags (the *rdf:Bag* keyword) and Lists (the *rdf:Seq* keyword). For denoting the i-th element of a container, the property *rdf:_i* is used. These keywords have no special meaning as-

[4]IRDFS2 and 3 do not appear in the normative RDFS document, but were noted to be necessary for preserving the RDFS entailment Lemma validity in [68]. "I" stands for "integrative".

sociated besides the axiomatic group of triples *(rdf:_i, rdf:type, rdfs:cmp), (rdf:_i, rdfs:domain, rdfs:Resource), (rdf:_i, rdfs:range, rdfs:Resource)*and *rdfs:cmp* is a shortcut for *rdfs:ContainerMembershipProperty*. The set A enforces that the i-th element of a container is of type resource and that each *rdf:_i* is a Container Membership Property. A is of infinite size, and thus requires some care when introduced in an actual implementation of an RDF(S) query system.

$\forall i \in \mathbb{N}$,
(rdf:_i, rdf:type, rdfs:ContainerMembershipProperty),
(rdf:_i, rdfs:domain, rdfs:Resource),
(rdf:_i, rdfs:range, rdfs:Resource)

Figure 3.2: The infinite portion of RDFS axiomatic triples A.

Finally, we say that a graph G_1 *rdfs-entails* a graph G_2 ($G_1 \models_s G_2$) if the RDFS closure $R(G_1)$ of G_1 entails G_2. For instance, if we add the triple (*Person*,*rdf:subClassOf*,*Animal*) to G_1', we have that, under RDFS semantics, the triple (*GB*,*rdf:type*,*Animal*), belongs to $R(G_1')$, by rule RDFS9 of Figure 3.3. This means that, if entailment is interpreted under RDFS semantics, other mappings, such as μ''' such that $\mu'''(\_b1) =$ *rdf:type* and $\mu'''(\_b2) =$ *Animal*, are a proof for entailment of G_2'' by G_1'.
In the following, we assume that $I \cup L \cup B \subseteq \mathcal{C}$. That is IRI, literals and blank nodes appearing in an RDF graph can be freely used as constant symbols in logic programs.

3.1.2 From RDF(S) to ASP^{HEX}

In order to encode in ASP the problem of deciding RDFS entailment between two graphs G_1 and G_2, we adopt two translations T_1 and T_2, and an ASP^{HEX} program D, which simulates RDFS entailment rules. T_1 transforms G_1 into a corresponding ASP^{HEX} program; intuitively, it transforms each triple $t \in G_1$ into a corresponding fact. T_1 maps blanks nodes to themselves, as it is conceptually done in the Skolemization Lemma of [80]. This transformation is intended to be applied on graphs which encode a dataset subject to querying, that is, the current reference ontology. We will see that, when a graph is seen as the description of a query pattern, we can adopt the second transformation T_2. T_2 transforms G_2 in a conjunction of atoms, whose truth values depend on whether G_2 is entailed by G_1 or not. T_2 maps blank nodes to variables.

Name	**If G contains:**	**then add to $R(G)$:**
RDF1	(u, a, y)	(a, rdf:type, rdf:Property)
RDF2A	(u, a, l) where l is a well-typed XML-literal	(l, rdf:type, rdf:XMLLiteral)
RDF2B	(u, a, l) where l is a ill-typed XML-literal	$(I \cup B \cup L) \times I \times (I \cup B \cup L)$
RDFS1A	(u, a, l) where l is a plain literal	(l, rdf:type, rdfs:Literal)
RDFS1B	(u, a, l) where l is a ill-typed literal	$(I \cup B \cup L) \times I \times (I \cup B \cup L)$
RDFS2	(a, rdfs:domain, x), (u, a, y)	(u, rdf:type, x)
RDFS3	(a, rdfs:range, x), (u, a, v)	(v, rdf:type, x)
IRDFS2	(a, rdfs:domain, b), (c,rdfs:subproperty,b), (x, c, y)	(x, rdf:type, b)
IRDFS3	(a, rdfs:range, b), (c,rdfs:subproperty,b), (x, c, y)	(y, rdf:type, b)
RDFS4A	(u, a, x)	(u, rdf:type, rdfs:Resource)
RDFS4B	(u, a, v)	(v, rdf:type, rdfs:Resource)
RDFS5	(u, rdfs:subPropertyOf, v), (v, rdfs:subPropertyOf, x)	(u, rdfs:subPropertyOf, x)
RDFS6	(u, rdf:type, rdf:Property)	(u, rdfs:subPropertyOf, u)
RDFS7	(a, rdfs:subPropertyOf, b), (u, a, y)	(u, b, y)
RDFS8	(u, rdf:type, rdfs:Class)	(u, rdfs:subClassOf, rdfs:Resource)
RDFS9	(u, rdfs:subClassOf, x), (v, rdf:type, u)	(v, rdf:type, x)
RDFS10	(u, rdf:type, rdfs:Class)	(u, rdfs:subClassOf, u)
RDFS11	(u, rdfs:subClassOf, v), (v, rdfs:subClassOf, x)	(u, rdfs:subClassOf, x)
RDFS12	(u, rdf:type, rdfs:ContainerMembershipProperty)	(u, rdfs:subPropertyOf, rdfs:member)
RDFS13	(u, rdf:type, rdfs:Datatype)	(u, rdfs:subClassOf, rdfs:Literal)

Figure 3.3: The RDF and RDFS Entailment rules for extending a graph G.

RDF1′	t(A,*rdf:type*,*rdf:Property*)	:–	t(U,A,Y).
RDF2A′	t(L,*rdf:type*,*rdfs:XMLLiteral*)	:–	t(U,A,L), *&XMLLiteral*[L](true).
RDF2B′		:–	t(L,*rdf:type*,*rdfs:XMLLiteral*), not *&XMLLiteral*[L](true).
RDFS1A′	t(L, *rdf:type*, *rdfs:Literal*)	:–	t(U,A,L), *&Literal*[L](true).
RDFS1B′		:–	t(L, *rdf:type*, *rdfs:Literal*), not *&Literal*[L](true).
RDFS2′	t(U,*rdf:type*,X)	:–	t(A,*rdfs:domain*,X), t(U,A,V).
RDFS3′	t(V,*rdf:type*,X)	:–	t(A,*rdfs:range*,X), t(U,A,V).
IRDFS2′	t(X,*rdf:type*B)	:–	t(A,*rdfs:domain*,B), t(X,C,Y), t(C,*rdfs:subproperty*,B).
IRDFS3′	t(Y,*rdf:type*B)	:–	t(A,*rdfs:range*,B), t(X,C,Y), t(C,*rdfs:subproperty*,B).
RDFS4A′	t(U,*rdf:type*,*rdfs:Resource*)	:–	t(U,A,X).
RDFS4B′	t(V,*rdf:type*,*rdfs:Resource*)	:–	t(U,A,V).
RDFS5′	t(U,*rdfs:subPropertyOf*,X)	:–	t(U,*rdfs:subPropertyOf*,V), t(V,*rdfs:subPropertyOf*,X).
RDFS6′	t(U,*rdfs:subPropertyOf*,U)	:–	t(U,*rdf:type*,*rdf:Property*).
RDFS7′	t(U,B,Y)	:–	t(A,*rdfs:subPropertyOf*,B), t(U,A,Y).
RDFS8′	t(U,*rdfs:subClassOf*, *rdfs:Resource*)	:–	t(U,*rdf:type*, *rdfs:Class*).
RDFS9′	t(V,*rdf:type*,X)	:–	t(U,*rdfs:subClassOf*,X), t(V,*rdf:type*,U).
RDFS10′	t(U,*rdfs:subClassof*,U)	:–	t(U,*rdf:type*,*rdfs:Class*).
RDFS11′	t(U,*rdfs:subClassof*,X)	:–	t(U,*rdfs:subClassOf*,V), t(V,*rdfs:subClassOf*,X).
RDFS12′	t(U,*rdfs:subPropertyOf*,*rdfs:member*)	:–	t(U,*rdf:type*,*rdfs:cmp*).
RDFS13′	t(U,*rdfs:subClassof*,*rdfs:Literal*)	:–	t(U,*rdf:type*,*rdfs:Datatype*).
A′	*symb*(X)	:–	t(X,Y,Z).
	symb(Y)	:–	t(X,Y,Z).
	symb(Z)	:–	t(X,Y,Z).
	CMProperty(P)	:–	*symb*(P), *&concat*[*"rdf:_"*,N](P).
	t(X, *rdf:type*, *rdfs:cmp*)	:–	*CMProperty*(X).
	t(X, *rdfs:domain*, *rdfs:Resource*)	:–	*CMProperty*(X).
	t(X, *rdfs:range*,*rdfs:Resource*)	:–	*CMProperty*(X).

Figure 3.4: Translation D of entailment rules shown in Figure 3.3 in ASP^{HEX}. $\&concat$ is defined as: $F_{\&concat}[$*"rdf:_"*$, N](P) = 1$ whenever P is the concatenation of the strings *"rdf:_"* and N. $\&Literal$ and $\&XMLLiteral$ are defined as follows: $F_{\&Literal}[X](true) = 1$ (resp. $F_{\&XMLLiteral}[X](true) = 1$) iff X is compliant with the syntax allowed for literals (resp. XML Literal syntax) [80]. *rdfs:cmp* is a shortcut for *rdfs:ContainerMembershipProperty*.

Definition 1 *The translation $T_1(G)$ of an RDF graph G is the set of facts $\{t(s,p,o) | (s,p,o) \in G\}$. The translation $T_2(G)$ of an RDF graph G is a program containing:*

- *the rule q = `entail` $\leftarrow H$. Here,* `entail` *is a fresh literal whereas H is the conjunction of all the atoms in the set $\{t(h(s), h(p), h(o)) \mid (s,p,o) \in G\}$, where h is a transformation function such that $h(a) = a$ if $a \in I \cup L$, $h(a) = A_a$ otherwise; A_a is a fresh variable associated with a if $a \in H$.*
- *the set of facts $\{symb(u) \mid u$ is an element of $I \cup L$ appearing in $G\}$.*

The program D, simulating RDFS entailment rules shown in Figure 3.3, is given in Figure 3.4. In particular, for each rule R in Figure 3.3 there is a counterpart R' in D (notably, RDF2B′ and RDFS1B′ are constraints rather than rules). The bottommost group of six rules in D (which we call A′) models a finite portion of the infinite set of triples A shown in Figure 3.2. The set of facts of the form $symb(u)$ denotes all the elements explicitly appearing in G_1 or in G_2.
Note that A' derives only a finite portion of the axiomatic triples: the axiomatic triples regarding a given identifier *rdf:_i* are derived if $symb($*rdf:_i*$)$ is true. In such a case *CMProperty*(rdf:_i) holds by means of the 4th rule of A', and this makes the head of the last but three rules of A' true.

In the following, let P be the ASP^{HEX} program $P = D \cup T_1(G_1) \cup T_2(G_2) \cup K$, for two given RDF graphs G_1 and G_2, where K is the set of facts

$\{symb($*rdf:_*$i_u) \mid u$ is a blank node appearing in G_2 and i_u is a distinguished natural number such that neither G_1 nor G_2 talk about *rdf:_*$i_u\}$.
The meaning of P in terms of its answer sets is given by the following Lemma. Roughly speaking the Lemma states that the unique answer set of P encodes in the extension of the predicate t a finite portion of the closure $R(G_1)$ of G_1. Namely, t contains only those triples of $R(G_1)$ that can be constructed using values belonging to S, where S is the set of constant symbols explicitly appearing in P. Note that S will contain all the symbols explicitly appearing in G_1 or G_2.

Lemma 3.1.1 *Let S be the set of constant symbols appearing in P. If P is consistent then its unique model M contains an atom $t(s,p,o)$ iff $(s,p,o) \in (S \times S \times S) \cap R(G_1)$.*

Proof. ($\Rightarrow$) If P is consistent, then its model M is clearly unique (P is a positive program except for the two constraints RDF2B′ and RDFS1B′). We focus now on atoms of the form $t(s,p,o)$ contained in M. Let a be one of such atoms. Observe that, by construction of P, s, p and o belong to

S. By observing the structure of P, we note that (s, p, o) corresponds to a triple of $R(G_1)$ for three possible reasons: either, *(i)* a corresponds to a fact in $T_1(G_1)$ (and thus $(s, p, o) \in G_1 \subseteq R(G_1)$), or *(ii)* a is the byproduct of the application of some of the rules from RDF1′ to RDFS13′ in D, except RDF2B′ and RDFS1B′ (and thus $(s, p, o) \in R(G_1)$), or *(iii)* a corresponds to an axiomatic triple modelled by the subprogram A' of D.

($\Leftarrow$). Consider a triple $a = (s, p, o) \in (S \times S \times S) \cap R(G_1)$. Clearly, a is either *(i)* a triple belonging to G_1 (and thus it has a corresponding atom $t(s, p, o) \in T_1(G_1)$), or, *(ii)* it is the byproduct of the application of some RDFS entailment rules, which are in one-to-one correspondence with rules RDF1′ to RDFS13′ in D, or *(iii)* it is an axiomatic triple. In the latter case, the subprogram A' guarantees that $a \in (S \times S \times S) \cap R(G_1) \Rightarrow t(s, p, o) \in M$.
$\square$

The following trivial Lemma, used in the subsequent Theorem 3.1.3, shows that if graphs do not refer explicitly to IRIs of the form $rdf{:}\_i$ (which are symbols appearing in the infinite set of axiomatic triples), then such IRIs are "interchangeable" in the sense explained by the Lemma, yet preserving RDFS entailment.

Lemma 3.1.2 **(Interchangeability Lemma**) *Given an RDF graph G, we say that G* talks *about an element $e \in I \cup L$ if there exists a triple $t \in G$ containing e. Let G_1 and G_2 be two RDF graphs for which there is μ such that $\mu(G_2) \subseteq R(G_1)$ (i.e. $G_1 \models_s G_2$).*
For each element $e \in I \cup B$ such that $\mu(e) = rdf{:}\_i$ ($i \in \mathbb{N}$), and such that G_1 and G_2 do not talk about e, then $\mu'(G_2) \subseteq R(G_1)$, for any μ' coinciding with μ except that $\mu'(e) =$ rdf:_i′ where $i' \neq i$ and G_1 and G_2 do not talk about rdf:_i′.

Theorem 3.1.3 *Let G_1 and G_2 be two RDF graphs. Then $G_1 \models_s G_2$ iff $P \models$* `entail`.

Proof. ($\Leftarrow$). Assume that $P \models$ `entail`. Observe that P is a stratified program not containing disjunctive rules, thus it may entail `entail` in two cases: *(a)* P is inconsistent. P can have no model only if G_1 is inconsistent, i.e. G_1 contains some literal which is not syntactically valid. This makes the body of either constraint RDFS1B′ or RDFS2B′ true (see Figure 3.4). Note that an inconsistent graph normatively entails any other graph by definition. *(b)* P contains `entail` in its unique answer set M. This means that the body H of the rule $q =$ `entail` $\leftarrow H$ has some ground instantiation which is modelled by M. By Lemma 3.1.1, it follows that $R(G_1) \models G_2$.

($\Rightarrow$). Given that $G_1 \models_s G_2$, then, if G_1 is inconsistent, P has no answer sets, thus trivially entailing the literal `entail`. If G_1 is consistent, there is a mapping μ from G_2 to $R(G_1)$. It is possible to show that the single

answer set M of P contains `entail`. For proving this, we need to show that $t(\mu(s), \mu(p), \mu(o)) \in M$ for each $(s, p, o) \in G_2$, thus showing that H has some ground instantiation which is true in M.
Consider $a = (s, p, o) \in G_2$. Clearly, given that $G_1 \models_s G_2$, a is such that $(\mu(s), \mu(p), \mu(o)) \in R(G_1)$. Let S be the set of symbols appearing in P as in Lemma 3.1.1. Note that, if $(\mu(s), \mu(p), \mu(o)) \in S \times S \times S$, then by Lemma 3.1.1 $t(\mu(s), \mu(p), \mu(o))$ belongs to M.
However, consider the set $V = \{v \text{ appears in } G_2 \mid \mu(v) \notin S\}$. Note that V might be nonempty and its cardinality not larger than $|K|$. In such a case, we cannot directly show that $t(\mu(s), \mu(p), \mu(o)) \in M$, since μ might map some element of G_2 to a value outside S. Also note that all the elements $v \in V$ are such that $\mu(v) =$ *rdf:_i* for some $i \in \mathbb{N}$, and that G_1 and G_2 do not talk about $\mu(v)$ (otherwise, it would be that $\mu(v) \in S$).
By Lemma 3.1.2, we can replace μ with a mapping μ' such that: *(i)* $\mu'(v) = v'$ and $v' =$ *rdf:_*i'_v, for all $v \in V$, where *i'* (and thus v') can be arbitrarily chosen; *(ii)* $\mu \equiv \mu'$ elsewhere. We also define μ' such that its restriction over V is a bijection to the elements of $K' = \{k \mid symb(k) \in K\}$. Note that μ' is a mapping to elements of S and is such that $\mu'(G_2) \subseteq R(G_1)$. By Lemma 3.1.1, $t(\mu'(s), \mu'(p), \mu'(o))$ belongs to M for any $(s, p, o) \in G_2$. □

Theorem 3.1.4 *Determining whether* $P \models$ `entail` *is decidable.*

Proof. Theorem 5 of [24], states that answer sets of a *vi-restricted* program P' can be computed by computing the answer set of a finite, ground program G'_P, and that G'_P is computable. P is *vi-restricted*. Then, entailment of `entail` can be decided in finite time[5]. □

3.2 Using ASP to query Semantic Web data

The notion of entailment and query answering are strictly related. Indeed, the notion of simple entailment (see 3.1.1) can be seen as a tool for formalizing query languages for RDF, by interpreting G_1 as the data to be queried, and G_2 as a query pattern that should be matched to G_1. Rules establishing matching criteria can be those of simple entailment (which basically consist in checking subgraph isomorphism between G_1 and G_2) or more complex ones. In this sense, entailment is decided by the existence

[5] P is an ordinary program, except for the fourth rule of A', containing possible value invention. Intuitively, P is a "safe" rule, in the sense that all the variables in its head appear also in a positive predicate in its body. This prevents actual value invention. For space reasons, we cannot include here details about vi-restrictedness. The interested reader is referred to [24].

of at least one mapping, while the set of matching mappings constitutes the possible answers to the the query pattern. Roughly speaking, when an entailment and/or a query answer has to be computed, a graph can have two different roles: it can play the role of the dataset to be queried, or that of the pattern representing the query at hand. For instance, the possible answers to the pattern graph $G_2'' = (\_b1, \textit{rdf:type}, \_b2)$ with respect to the graph G_1' in Figure 3.1, are the two mappings μ' and μ'' where $\mu'(\_b1) =$ *GB*, $\mu'(\_b2) =$ *Person*, and $\mu''(\_b1) =$ *Axel*, $\mu''(\_b2) =$ *Person*. Both μ' and μ'' testify that G_1' entails G_2''.
In practice, this means that if one wants to use G_2 for querying information about G_1 (which is expected to correspond to a large set of triples) under RDFS entailment, G_2 is actually matched to $R(G_1)$, which contains also inferred knowledge.
Therefore, based on a previous result showing that the SPARQL query language can be mapped to a rule-based language ([81]), with stable model semantics, we show that query answering can be accomplished using the ASP paradigm. In practise, we show that efficient querying of ontologies can be accomplished with proper extensions of the well known ASP system DLV.

3.2.1 Querying RDFS ontologies using ASP

By adapting the encoding obtained in the section 3.1.2 the way is paved to an actual implementation of an RDF query system based on ASP. In particular, given the graph G_1 (the dataset to be queried) and a graph G_2 (a query pattern), and considering the program $P = T_1(G_1) \cup T_2(G_2) \cup D$:

• query answering amounts to finding all the possibilities to match G_2 to G_1. In our case, it is enough to modify the rule `entail` $\leftarrow H$ in $T_2(G_2)$, by changing `entail` to an atom $answer(X_1, \ldots, X_n)$ where variables $X_1, \ldots, X_n$ correspond to a subset of choice of blank nodes in G_2[6].

• the core of the SPARQL language is a syntactic method for expressing G_2 and the variables exposed in the predicate *answer*. A basic graph pattern written in SPARQL represents indeed a query pattern like G_2, which is converted to $T_2(G_2)$. Further, more involved features of SPARQL can be accommodated in our encoding by changing the transformation T_2, as explained in [81]. The same holds also for query patterns expressed in other non-standard query languages.

[6]As a simple Corollary of Theorem 3.1.3, we note that under RDFS entailment, the predicate *answer* encodes a finite subset F of all the possible answers Ans. If Ans is finite, then $F = Ans$.

• D can be easily customized to the entailment regime of interest. For instance, if one is interested in ρDF entailment only, it is sufficient to remove from D all the rules except RDF1′, RDFS2 and 3′, IRDF2′ and 3′, RDFS5′,6′,7′,8′,9′,10′ and 11′.

Chapter 4

Enhancing SPARQL: Dynamic Querying and Rule-Based Entailment Regimes

Generic extensions for SPARQL to entailment regimes other than simple RDF entailment are still an open research problem,[1] due to various problems: (i) for (non-simple) RDF entailment regimes, such as full RDFS entailment, the deductive closure of the original graph is infinite, and thus SPARQL queries over an empty graph might already have infinite answers, and (ii) it is not yet clear which should be the intuitive answers to queries over inconsistent graphs, e.g. in OWL entailment, etc. In fact, SPARQL restricts extensions of basic graph pattern matching to retain finite answers.

Moreover, we show that SPARQL faces certain unwanted ramifications when querying ontologies under RDFS entailment regimes in conjunction with RDF datasets that comprise multiple named graphs, and we provide an extension for SPARQL that remedies these effects.

Finally, dynamically assigning different ontologies or rulesets to data for querying is neither supported by the SPARQL specification nor by existing systems. Most available RDF databases only offer limited support for dynamic inference, custom reasoning via rules, or querying upon datasets of choice.

Therefore, we propose an ASP based solution that allows for dynamically choosing the set of entailment rules to be adopted for the inference at query time, enabling SPARQL querying on varying datasets and varying schemes as well as supporting rule-based RDFS and higher inference.

[1]For details, cf. `http://www.polleres.net/sparqltutorial/`, Unit 5b.

```
@prefix foaf:  <http://xmlns.com/foaf/0.1/>.
@prefix rel:  <http://purl.org/vocab/relationship/>.
...
rel:friendOf rdfs:subPropertyOf foaf:knows.
foaf:knows rdfs:domain foaf:Person.
foaf:knows rdfs:range foaf:Person.
foaf:homepage rdf:type owl:inverseFunctionalProperty.
...
```

(a) Graph G_M (`<http://example.org/myOnt.rdfs>`), a combination of the FOAF&Relationship ontologies.

```
<http://bob.org#me> foaf:name "Bob";
a foaf:Person;
   foaf:homepage
<http://bob.org/home.html>;
   rel:friendOf [ foaf:name "Alice";
                  rdfs:seeAlso
<http://alice.org> ].
```

(b) Graph G_B (`<http://bob.org>`)

```
<http://alice.org#me> foaf:name
"Alice"; a foaf:Person;
rel:friendOf [ foaf:name "Charles" ],
             [ foaf:name "Bob";
               foaf:homepage
<http://bob.org/home.html> ].
```

(c) Graph G_A (`<http://alice.org>`)

Figure 4.1: An ontology and two data graphs

4.1 Motivation and Examples

Let us just start right away with some illustrating example motivating our proposed extensions of SPARQL; we assume two data graphs describing data about our well-known friends Bob and Alice shown in Fig. 4.1(b)+(c). Both graphs refer to terms in a combined ontology defining the FOAF and Relationship[2] vocabularies, see Fig. 4.1(a) for an excerpt.
On this data the SPARQL query (4.1) intends to extract names of persons mentioned in those graphs that belong to friends of Bob. We assume that, by means of `rdfs:seeAlso` statements, Bob provides links to the graphs associated to the persons he is friend with. Here, the **`from`** and **`from named`** clauses specify an RDF dataset.

```
select ?N from <http://example.org/myOnt.rdfs>
          from <http://bob.org>
          from named <http://alice.org>                              (4.1)
where { <http://bob.org#me> foaf:knows ?X . ?X rdfs:seeAlso ?G .
        graph ?G { ?P rdf:type foaf:Person; foaf:name ?N } }
```

In particular, the dataset of query (4.1) would be defined as $DS_1 = (\ G_M \uplus G_B, \{(\texttt{<http://alice.org>}, G_A)\})$, where $\uplus$ denotes merging of graphs according to the normative specifications.
Now, let us have a look at the answers to query (4.1). Answers to SPARQL **`select`** queries are defined in terms of multisets of partial variable substitutions. In fact the answer to query (4.1) is empty when – as typical for current SPARQL engines – only simple RDF entailment is taken into account,

[2]`http://vocab.org/relationship/.`

and query answering then boils down to simple graph matching. Since neither of the graphs in the default graph contain any triple matching the pattern `<http://bob.org#me> foaf:knows ?X` in the **`where`** clause, the result of (4.1) is empty. When taking subproperty inference by the statements of the ontology in G_M into account, however, one would expect to obtain three substitutions for the variable ?N: {?N/"Alice", ?N/"Bob", ?N/"Charles"}. We will explain in the following why this is not the case in standard SPARQL.
In order to obtain the expected answer, firstly SPARQL's basic graph pattern matching needs to be extended, see [85, Section 12.6]. In theory, this means that the graph patterns in the **`where`** clause needs to be matched against an enlarged version of the original graphs in the dataset (the deductive closure) of a given entailment regime. Not surprisingly, many existing implementations implement finite approximations of higher entailment regimes such as RDFS and OWL ([74, 102, 59]). E.g., the RDF Semantics document [80] contains an informative set of entailment rules, a subset of which (such as the one presented in Section 4.4 below) is implemented by most available RDF stores. These rule-based approximations, are typically expressible by means of Datalog-style rules. These latter model how to infer a finite closure of a given RDF graph that covers sound but not necessarily complete RDF(S) and OWL inferences. It is worth noting that rule-based entailment can be implemented in different ways: rules could be either dynamically evaluated upon query time, or the closure of a graph wrt. ruleset could be materialized when the graph is loaded into a store.

Materialization of inferred triples at loading time allows faster query responses, yet it has drawbacks: it is time and space expensive and it has to be performed once and statically. In this setting, it must be decided upfront *(i)*which ontology should be taken into account for which data graph, and *(ii)* to which graph(s) the inferred triples "belong", which particularly complicates the querying of named graphs.
As for exemplifying *(i)*, assume that a user agent wants to issue another query on graph G_B with only the FOAF ontology in mind, since she does not trust the Relationship ontology. In the realm of FOAF, `rel:friendOf` has nothing to deal with `foaf:knows`. However, when materializing all inferences upon loading G_M and G_B into the store, `bob:me foaf:knows _:a` would be inferred from $G_M \uplus G_B$ and would contribute to such a different query. Current RDF stores prevent to dynamically parameterize inference with an ontology of choice at query time, since indeed typically all inferences are computed upon loading time *once and for all*.
As for *(ii)*, queries upon datasets including named graphs are even more problematic. Query (4.1) uses G_B in order to find the IRI identifiers for

persons that Bob knows by following `rdfs:seeAlso` links and looks for persons and their names in the *named* RDF graphs found at these links. Even if rule-based inference was supported, the answer to query (4.1) over dataset DS_1 is just $\{?N/$"`Alice`"$\}$, as "Alice" is the only (explicitly) asserted `foaf:Person` in G_A. Subproperty, domain and range inferences over the G_M ontology do not propagate to G_A, since G_M is normatively prescribed to be merged into the default graph, but not into the named graph. Thus, there is no way to infer that "`Bob`" and "`Charles`" are actually names of `foaf:Person`s within the named graph G_A. Indeed, SPARQL does not allow to merge, on demand, graphs into the named graphs, thus there is no way of combining G_M with the named graph G_A.

4.2 SPARQL Graph pattern matching

Preliminaries. In the following, we assume I, B, and L denoting pairwise disjoint infinite sets of IRIs, blank nodes, and RDF literals, respectively. A *term* is an element from $I \cup B \cup L$. An *RDF graph* G (or simply *graph*) is defined as a set of *triples* from $I \cup B \times I \cup B \times I \cup B \cup L$ (cf. [79, 9]); by $blank(G)$ we denote the set of blank nodes of G.[3]
A *blank node renaming* θ is a mapping $I \cup B \cup L \rightarrow I \cup B \cup L$. We denote by $t\theta$ the application of θ to a term t. If $t \in I \cup L$ then $t\theta = t$, and if $t \in B$ then $t\theta \in B$. If (s, p, o) is a triple then $(s, p, o)\theta$ is the triple $(s\theta, p\theta, o\theta)$. Given a graph G, we denote by $G\theta$ the set of all triples $\{t\theta \mid t \in G\}$. Let G and H be graphs. Let θ_H^G be an arbitrary blank node renaming such that $blank(G) \cap blank(H\theta_H^G) = \emptyset$. The *merge of G by H*, denoted $G \uplus H$, is defined as $G \cup H\theta_H^G$.

SPARQL introduces the notion of RDF dataset, which is the pairing of a default graph and zero or more named graphs. Formally, an *RDF dataset* $D = (G_0, N)$ is a pair consisting of exactly one unnamed graph, the so-called default graph G_0, and a set $N = \{\langle u_1, G_1\rangle, \ldots, \langle u_n, G_n\rangle\}$ of named graphs, coupled with their identifying URIs. The following conditions hold: (i) each G_i $(0 \leq i \leq n)$ is a graph, (ii) each u_j $(1 \leq j \leq n)$ is from the set of IRIs I, and (iii) for all $i \neq j$, $\langle u_i, G_i\rangle, \langle u_j, G_j\rangle \in N$ implies $u_i \neq u_j$ and $blank(G_i) \cap blank(G_j) = \emptyset$.In the following, we restrict ourselves to **`select`** queries as shown in the example queries (4.1)–(4.4) and just provide an overview of the necessary concepts.

[3]Note that we allow *generalized RDF graphs* that may have blank nodes in property position.

A query in SPARQL can be viewed as a tuple $Q = (V, D, P)$, where V is the set of variables mentioned in the **select** clause, D is an RDF dataset, defined by means of **from** and **from named** clauses, and P is a *graph pattern*, defined in the **where** clause. Graph patterns are in the simplest case sets of RDF triples (s, p, o), where terms and variables from an infinite set of variables Var are allowed, also called *basic graph patterns (BGP)*. More complex graph patterns can be defined recursively, i.e., if P_1 and P_2 are graph patterns, $g \in I \cup Var$ and R is a filter expression, then P_1 **optional** P_2, P_1 **union** P_2, P_1 **filter** R, and **graph** $g\, P_1$ are graph patterns.
Queries are evaluated by matching graph patterns against graphs in the dataset. In order to determine a query's solution, in the simplest case BGPs are matched against the *active graph* of the query, which is one particular graph in the dataset, identified as shown next. Solutions of BGP matching consist of multisets of bindings for the variables mentioned in the pattern to terms in the active graph.

Partial solutions of each subpattern are joined according to an algebra defining the **optional**, **union** and **filter** operators, cf. [85, 79, 9]. For what we are concerned with here, the most interesting operator though is the **graph** operator, since it changes the active graph. That is, the active graph is the default graph G_0 for any basic graph pattern not occurring within a **graph** sub pattern. However, in a subpattern **graph** $g\,\{P_1\}$, the pattern P_1 is matched against the named graph identified by g, if $g \in I$, and against any named graph u_i, if $g \in Var$, where the binding u_i is returned for variable g. According to [9], for a RDF dataset D and active graph G, we define $[\![P]\!]^D_G$ as the multiset of tuples constituting the *answer* to the graph pattern P.
The solutions of a query $Q = (V, D, P)$ is the projection of $[\![P]\!]^D_G$ to the variables in V only.

4.3 SPARQL with Extended Datasets

To remedy the deficiencies outlined in section 4.1, we suggest an extension of the SPARQL syntax, in order to allow the specification of datasets more flexibly: it is possible to group graphs to be merged in parentheses in **from** and **from named** clauses. The modified query, obtaining a dataset $DS_2 = (G_M \uplus G_B, \{(\texttt{http://alice.org}, G_M \uplus G_A)\})$ looks as follows:

```
select ?N
  from (<http://example.org/myOnt.rdfs> <http://bob.org/>)
  from named <http://alice.org>
     (<http://example.org/myOnt.rdfs> <http://alice.org/>)
where { bob:me foaf:knows ?X . ?X rdfs:seeAlso ?G .
        graph ?G { ?X foaf:name ?N . ?X a foaf:Person . } }
```
(4.2)

For ontologies which should apply to the whole query, i.e., graphs to be merged into the default graph as well as any specified named graph, we suggest a more convenient shortcut notation by adding the keyword **`using ontology`** in the SPARQL syntax:

```
select ?N
  using ontology <http://example.org/myOnt.rdfs>
  from <http://bob.org/>
  from named <http://alice.org/>                              (4.3)
where { bob:me foaf:knows ?X . ?X foaf:seeAlso ?G .
        graph ?G { ?X foaf:name ?N . ?X a foaf:Person. } }
```

Hence, the **`using ontology`** construct allows for coupling the entire given dataset with the terminological knowledge in the `myOnt` data schema. As our investigation of currently available RDF stores shows, none of these systems easily allow to merge ontologies into named graphs or to dynamically specify the dataset of choice.

4.3.1 Parameterizing queries with ontologies

In the following we precisely define the SPARQL extensions outlined above, formalizing the notion of *dynamic querying* in terms of the dependence of BGP pattern answers $[\![P]\!]^{\mathcal{O},\mathcal{R}}$ from a variable ontology $\mathcal{O}$. For our exposition, we rely on well-known definitions of RDF datasets and SPARQL. What is important to note now is that, by the way how datasets are syntactically defined in SPARQL, the default graph G_0 can be obtained from merging a group of different source graphs, specified via several **`from`** clauses – as shown, e.g., in query (4.1) – whereas in each **`from named`** clause a single, separated, named graph is added to the dataset. That is, **`graph`** patterns will always be matched against a separate graph only.
To generalize this approach towards dynamic construction of groups of merged named graphs, we introduce the notion of an *extended dataset*, which can be specified by enlarging the syntax of SPARQL with two additional dataset clauses:

- For $i, i_1, \ldots, i_m$ distinct IRIs ($m \geq 1$), the statement"**`from named`** $i(i_1 \ldots i_m)$" is called *extended dataset clause*. Intuitively, $i_1 \ldots i_m$ constitute a group of graphs to be merged: the merged graph is given i as identifying IRI.
- For $o \in I$ we call the statement "**`using ontology`** o" an *ontological dataset clause*. Intuitively, o stands for a graph that will merged with all graphs in a given query.

Extended RDF datasets are thus defined as follows. A *graph collection* $\mathcal{G}$ is a set of RDF graphs.

An *extended RDF dataset* $\mathcal{D}$ is a pair $(\mathcal{G}_0, \{\langle u_1, \mathcal{G}_1\rangle, \ldots, \langle u_n, \mathcal{G}_n\rangle\})$ satisfying the following conditions: (i) each $\mathcal{G}_i$ is a nonempty graph collection (note that $\{\emptyset\}$ is a valid nonempty graph collection), (ii) each u_j is from I, and (iii) for all $i \neq j$, $\langle u_i, \mathcal{G}_i\rangle, \langle u_j, \mathcal{G}_j\rangle \in D$ implies $u_i \neq u_j$ and for $G \in \mathcal{G}_i$ and $H \in \mathcal{G}_j$, $blank(G) \cap blank(H) = \emptyset$. We denote $\mathcal{G}_0$ as $dg(\mathcal{D})$, the *default graph collection* of $\mathcal{D}$.

Let $\mathcal{D}$ and $\mathcal{O}$ be an extended dataset and a graph collection, resp. The ordinary RDF dataset obtained from $\mathcal{D}$ and $\mathcal{O}$, denoted $D(\mathcal{D}, \mathcal{O})$, is defined as

$$\left(\biguplus_{g \in dg(\mathcal{D})} g \uplus \biguplus_{o \in \mathcal{O}} o, \left\{ \langle u, \biguplus_{g \in \mathcal{G}} g \uplus \biguplus_{o \in \mathcal{O}} o\rangle \mid \langle u, \mathcal{G}\rangle \in \mathcal{D} \right\} \right) .$$

We can now define the semantics of extended and ontological dataset clauses as follows. Let F be a set of ordinary and extended dataset clauses, and O be a set of ontological dataset clauses. Let $graph(g)$ be the graph associated to the IRI g: the extended RDF dataset obtained from F, denoted $edataset(F)$, is composed of:

(1) $\mathcal{G}_0 = \{graph(g) \mid$ "**from** g" $\in F\}$. If there is no **from** clause, then $\mathcal{G}_0 = \emptyset$.

(2) A named graph collection $\langle u, \{graph(u)\}\rangle$ for each "**from named** u" in F.

(3) A named graph collection $\langle i, \{graph(i_1), \ldots, graph(i_m)\}\rangle$ for each "**from named** $i(i_1 \ldots i_m)$" in F.

The graph collection obtained from O, denoted $ocollection(O)$, is the set $\{graph(o) \mid$ "**using ontology** o" $\in O\}$. The ordinary dataset of O and F, denoted $dataset(F, O)$, is the set $D(edataset(F), ocollection(O))$.

Let $\mathcal{D}$ and $\mathcal{O}$ be as above. The *evaluation* of a graph pattern P over $\mathcal{D}$ and $\mathcal{O}$ having active graph collection $\mathcal{G}$, denoted $[\![P]\!]_{\mathcal{G}}^{\mathcal{D},\mathcal{O}}$, is the evaluation of P over $D(\mathcal{D}, \mathcal{O})$ having active graph $G = \biguplus_{g \in \mathcal{G}} g$, that is, $[\![P]\!]_{\mathcal{G}}^{\mathcal{D},\mathcal{O}} = [\![P]\!]_{G}^{D(\mathcal{D},\mathcal{O})}$.

Note that the semantics of extended datasets is defined in terms of ordinary RDF datasets. This allows to define the semantics of SPARQL with extended and ontological dataset clauses by means of the standard SPARQL semantics. Also note that our extension is conservative, i.e., the semantics coincides with the standard SPARQL semantics whenever no ontological clauses and extended dataset clauses are specified.

4.4 SPARQL with Arbitrary Rulesets

Since RDFS inference has a close relationship with logic rules, we generalize our approach to select a custom ruleset for specifying inferences to be taken into account in a SPARQL query. In addition to parameterizing queries with ontologies in the dataset clauses, we allow to parameterize the ruleset which models the entailment regime at hand.

Per default, our framework supports a standard ruleset that emulates (a finite subset of) the RDFS semantics. Alternatively, different rule-based entailment regimes, e.g., rulesets covering parts of the OWL semantics á la ter Horst [102], de Bruijn [28, Section 9.3], OWL2 RL [71] or other custom rulesets can be referenced with the **using ruleset** keyword. As an example, the following query returns the solution{?X/<http://alice.org#me>}, ?Y/<http://bob.org#me>}, by doing equality reasoning over inverse functional properties such as `foaf:homepage` when the FOAF ontology is being considered:

```
select ?X ?Y
 using ontology <http://example.org/myOnt.rdfs>
 using ruleset rdfs
 using ruleset <http://www.example.com/owl-horst>                    (4.4)
 from <http://bob.org/>
 from <http://alice.org/>
where { ?X foaf:knows ?Y }
```

Query (4.4) uses the built-in RDFS rules for the usual subproperty inference, plus a ruleset implementing ter Horst's inference rules, which might be available at URL `http://www.example.com/owl-horst`. This ruleset contains the following additional rules, that will "equate" the blank node used in G_A for "Bob" with <http://bob.org#me>:[4]

```
?P rdf:type owl:iFP . ?S1 ?P ?O . ?S2 ?P ?O .    → ?S1 owl:sameAs ?S2.
?X owl:sameAs ?Y                                 → ?Y owl:sameAs ?X.
?X ?P ?O . ?X owl:sameAs ?Y                      → ?Y ?P ?O.          (4.5)
?S ?X ?O . ?X owl:sameAs ?Y                      → ?S ?Y ?O.
?S ?P ?X . ?X owl:sameAs ?Y                      → ?S ?P ?Y.
```

4.4.1 Parameterizing queries with ruleset

In the following we define the notion of *dynamic querying* formalized in terms of the dependence of BGP pattern answers $[\![P]\!]^{\mathcal{O},\mathcal{R}}$ from a variable ruleset $\mathcal{R}$. For our exposition, we rely on well-known definitions of RDF datasets and SPARQL.

[4]We use `owl:iFP` as shortcut for `owl:inverseFunctionalProperty`.

Extended dataset clauses give the possibility of merging arbitrary ontologies into any graph in the dataset. The second extension herein presented enables the possibility of dynamically deploying and specifying rule-based entailments regimes on a per query basis. To this end, we define a generic $\mathcal{R}$-entailment, that is RDF entailment associated to a parametric ruleset $\mathcal{R}$ which is taken into account when evaluating queries. For each such $\mathcal{R}$-entailment regime we straightforwardly extend BGP matching, in accordance with the conditions for such extensions as defined in [85, Section 12.6].

We define an *RDF inference rule* r as the pair $(\mathcal{A}nte, \mathcal{C}on)$, where the *antecedent* $\mathcal{A}nte$ and the *consequent* $\mathcal{C}on$ are basic graph patterns such that $\mathcal{V}(\mathcal{C}on)$ and $\mathcal{V}(\mathcal{A}nte)$ are non-empty, $\mathcal{V}(\mathcal{C}on) \subseteq \mathcal{V}(\mathcal{A}nte)$ and $\mathcal{C}on$ does not contain blank nodes.[5] As in Example (4.5) above, we typically write RDF inference rules as

$$\mathcal{A}nte \rightarrow \mathcal{C}on \ . \tag{4.6}$$

We call sets of inference rules *RDF inference rulesets*, or *rulesets* for short.

Rule Application and Closure. We define *RDF rule application* in terms of the immediate consequences of a rule r or a ruleset $\mathcal{R}$ on a graph G. Given a BGP P, we denote as $\mu(P)$ a pattern obtained by substituting variables in P with elements of $I \cup B \cup L$. Let r be a rule of the form (4.6) and G be a set of RDF triples, then:

$$T_r(G) = \{\mu(\mathcal{C}on) \mid \exists\mu \text{ such that } \mu(\mathcal{A}nte) \subseteq G\}.$$

Accordingly, let $T_{\mathcal{R}}(G) = \bigcup_{r\in\mathcal{R}} T_r(G)$. Also, let G_0 = G and $G_{i+1} = G_i \cup T_{\mathcal{R}}(G_i)$ for $i \geq 0$. It can be easily shown that there exists the smallest n such that $G_{n+1} = G_n$; we call then $Cl_{\mathcal{R}}(G) = G_n$ the *closure* of G with respect to ruleset $\mathcal{R}$.

We can now further define *$\mathcal{R}$-entailment* between two graphs G_1 and G_2, written $G_1 \models_{\mathcal{R}} G_2$, as $Cl_{\mathcal{R}}(G_1) \models G_2$. Obviously for any finite graph G, $Cl_{\mathcal{R}}(G)$ is finite. In order to define the semantics of a SPARQL query wrt. *$\mathcal{R}$-entailment* we now extend graph pattern matching in $[\![P]\!]^D_G$ towards respecting $\mathcal{R}$.

Definition 2 (extended basic graph pattern matching for $\mathcal{R}$-entailment) *Let D be a dataset and G be an active graph. The* solution of a BGP P wrt. $\mathcal{R}$-entailment, *denoted $[\![P]\!]^{D,\mathcal{R}}_G$, is $[\![P]\!]^D_{Cl_{\mathcal{R}}(G)}$.*

[5]Unlike some other rule languages for RDF, the most prominent of which being CONSTRUCT statements in SPARQL itself, we forbid blank nodes; i.e., existential variables in rule consequents which require the "invention" of new blank nodes, typically causing termination issues.

The solution $[\![P]\!]_G^{D,\mathcal{R}}$ naturally extends to more complex patterns according to the SPARQL algebra. In the following we will assume that $[\![P]\!]_G^{D,\mathcal{R}}$ is used for graph pattern matching. Our extension of basic graph pattern matching is in accordance with the conditions for extending BGP matching in [85, Section 12.6]. Basically, these conditions say that any extension needs to guarantee finiteness of the answers, and defines some conditions about a "*scoping graph*." Intuitively, for our extension, the scoping graph is just equivalent to $Cl_{\mathcal{R}}(G)$. We refer to [85, Section 12.6] for the details. To account for this generic SPARQL BGP matching extension parameterized by an RDF inference ruleset $\mathcal{R}_Q$ per SPARQL query Q, we introduce another novel language construct for SPARQL:

- For $r \in I$ we call "**`using ruleset`** r" a *ruleset clause*.

Analogously to IRIs denoting graphs, we now assume that an IRI $r \in I$ may not only refer to graphs but also to rulesets, and denote the corresponding ruleset by $ruleset(r)$. Each query Q may contain zero or more ruleset clauses, and we define the query ruleset $\mathcal{R}_Q = \bigcup_{r \in R} ruleset(r)$, where R is the set of all ruleset clauses in Q.
The definitions of solutions of a query and the evaluation of a pattern in this query on active graph G is now defined just as above, with the only difference that answer to a pattern P are given by $[\![P]\!]_G^{D,\mathcal{R}_Q}$. We observe that whenever $\mathcal{R} = \emptyset$, then $\mathcal{R}$-entailment boils down to simple RDF entailment. Thus, a query without ruleset clauses will just be evaluated using standard BGP matching. In general, our extension preserve full backward compatibility.

Proposition 1 *For $\mathcal{R} = \emptyset$ and RDF graph G,* $[\![P]\!]_G^{D,\mathcal{R}} = [\![P]\!]_G^D$.

Analogously, one might use *$\mathcal{R}$-entailment* as the basis for RDFS entailment as follows. We consider here the ρDF fragment of RDFS entailment [74]. Let $\mathcal{R}_{RDFS}$ denote the ruleset corresponding to the minimal set of entailment rules (2)–(4) from [**?**]:

```
?P rdfs:subPropertyOf ?Q .?Q rdfs:subPropertyOf ?R .    → ?P rdfs:subPropertyOf ?R.
?P rdfs:subPropertyOf ?Q . ?S ?P ?O .                   → ?S ?Q ?O.
?C rdfs:subClassOf ?D . ?D rdfs:subClassOf ?E .         → ?C rdfs:subClassOf ?E.
?C rdfs:subClassOf ?D . ?S rdf:type ?C .                → ?S rdf:type ?D.
?P rdfs:domain ?C . ?S ?P ?O .                          → ?S rdf:type ?C.
?P rdfs:range ?C . ?S ?P ?O .                           → ?O rdf:type ?C.
```

Since obviously $G \models_{RDFS} Cl_{\mathcal{R}_{RDFS}}(\mathcal{G})$ and hence $Cl_{\mathcal{R}_{RDFS}}(\mathcal{G})$ may be viewed as a finite approximation of RDFS-entailment, we can obtain a reasonable definition for defining a BGP matching extension for RDFS by simply defining $[\![P]\!]_G^{D,RDFS} = [\![P]\!]_G^{D,\mathcal{R}_{RDFS}}$. We allow the special ruleset clause

`using ruleset` `rdfs` to conveniently refer to this particular ruleset. Other rulesets may be published under a Web dereferenceable URI, e.g., using an appropriate RIF [18] syntax. Note, eventually, that our rulesets consist of positive rules, and as such enjoy a natural monotonicity property.

Proposition 2 *For rulesets $\mathcal{R}$ and $\mathcal{R}'$, such that $\mathcal{R} \subseteq \mathcal{R}'$, and graph G_1 and G_2, if $G_1 \models_{\mathcal{R}} G_2$ then $G_1 \models_{\mathcal{R}'} G_2$.*

Entailment regimes modeled using rulesets can thus be enlarged without retracting former inferences. This for instance would allow to introduce tighter RDFS-entailment approximations by extending $\mathcal{R}_{RDFS}$ with further axioms, yet preserving inferred triples.

Chapter 5

GiaBATA: a novel Framework for the Semantic Web

This chapter intents to provide the technical feasibility of our approach presenting a system, called GiaBATA for storing, aggregating, and querying Semantic Web data. It is based on Answer Set Programming technology, namely on the DLVHEX system, which allows us to implement a fully SPARQL compliant semantics, and on DLV^{DB}, which extends the DLV system with persistent storage capabilities. Our proposed architecture provides a fully declarative implementation of SPARQL and brings support for further (deductive) database and logic programming optimization and extensibility.
Moreover, the system allows for extensions of SPARQL by non-standard features such as aggregates, custom built-ins, or arbitrary rulesets, providing a flexible toolbox that embeds Semantic Web data and ontologies in a fully declarative logic programming environment.

5.1 Architectural Overview

Traditional RDF processors are designed for handling large RDF graphs in memory, thus reaching their limits very early when dealing with large graphs retrieved from the Web. Current RDF Stores, such as YARS [54], Sesame, Jena TDB, ThreeStore, AllegroGraph, or OpenLink Virtuoso[1] provide roughly the same functionality as traditional relational database systems do for relational data. They offer query facilities and allow to im-

[1]See `http://openrdf.org/`, `http://jena.hpl.hp.com/wiki/TDB/`, `http://threestore.sf.net/`, `http://agraph.franz.com/allegrograph/`, `http://openlinksw.com/virtuoso/`, respectively.

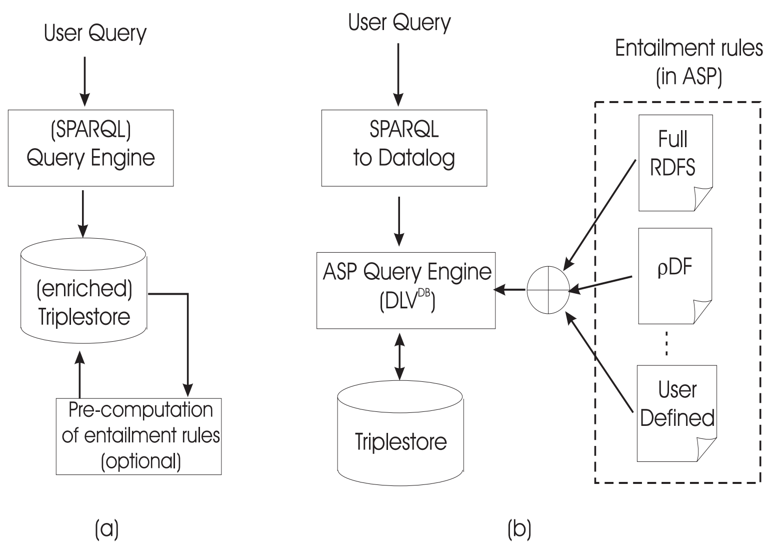

Figure 5.1: *(a)* Traditional architectures for querying RDF(S) triplestores. *(b)* Our proposal.

port large amounts of RDF data into their persistent storage, and typically support SPARQL [85]. Figure 5.1(a) shows the generic architecture of most current triplestore querying systems. In particular, the triplestore acts as a database and the query engine (possibly a SPARQL-enabled one) manipulates this data. Most of the current query engines adopt a *pre-materialization* approach, where entailment rules are pre-computed and the initial triplestore is enriched with their consequences before carrying out any querying activity.
Unfortunately, triplestores based on the pre-materialization approach outlined above have some drawbacks:

• Inferred information is available only after the often long lasting pre-materialization step. Pre-materialization is unpractical if massive amounts of data are involved in the inferencing process; in fact, inferred information is usually much bigger in size than the original one. As an example, if only the ρDF fragment of RDFS is considered, this growth has been empirically estimated (in, e.g., [96]) as more than twice the original size of the dataset. Actually, a materialized dataset can be cubically larger in theory.

• Entailment rules are “statically” programmed by coupling a parametric reasoner (designed “ad-hoc”) with the original triplestore code. This prevents the possibility to dynamically prototype new inference rules, and to activate/de-activate inference rules depending on the given application. For instance, one might want to restrict RDF(S) inference only to the known ρDF fragment, or to enlarge inference with other (normative or not) entailment rules such as the D-entailment rules introduced in [80].

• The basic reasoning machinery of RDF(S) prescribes a heavy usage of transitive closure (recursive) constructs. Roughly speaking, given a class taxonomy, an individual belonging to a leaf class must be inferred to be member of all the ancestor classes, up to the root class. This prevents a straightforward implementation of RDF(S) over RDFBMs, since RDFBMs usually feature very primitive, and inefficient, implementations of recursion in their native query languages.

In our approach the user can choose on the fly, at query time, which set of entailment rules must be applied on the triplestore, and/or design his own. Note that, rather than materializing the whole output of the application of entailment rules, the rules are used as an inference mechanism for properly answering the specific input query at hand.

The framework has been implemented by reducing queries, datasets and rulesets to a common ground which allows arbitrary interoperability between the three realms. This common ground is ASP, wherein rulesets naturally fit and SPARQL queries can be reduced to. Subsequently, the resulting combined ASP programs can be evaluated over an efficient SQL interface to an underlying relational DBMS that works as triple store.
A key issue in is whether it could be possible to achieve an efficient interaction of the ASP engine with the database. In fact, it is well known that current (extended) Datalog-based systems present important limitations when the amount of data to reason about is large: *(i)* reasoning is generally carried out in main-memory and, hence, the quantity of data that can be handled simultaneously is limited; *(ii)* the interaction with external (and independent) DBMSs is not trivial and, in several cases, not allowed at all, but in order to effectively share and elaborate large ontologies these must be handled with some database technology; *(iii)* the efficiency of present Datalog evaluators is still not sufficient for their utilization in complex reasoning tasks involving large amounts of data. We solved this issue implementing the query engine by means of a database oriented version of the DLV solver for Answer Set Programming programs, which can both access and modify data in the underlying database.

5.1.1 Components of the System

Fig. 5.2 shows a high-level view of our system architecture. It consists of four main components:

• a SPARQL to Datalog rewriter as part of a plugin for DLVHEX (see [81, 84]),

• a schema rewriter which manipulates the rewritten datalog rules, adds auxiliary definitions in order to match the underlying database schema, and implements the chosen semantics by adding rules to the input of the module in,

• the DLV^{DB} Answer Set Programming engine, which rewrites the input program to native SQL queries, and thereby accesses triples persistently stored in

• a DBMS of choice storing the RDF data according to a particular storage schema.

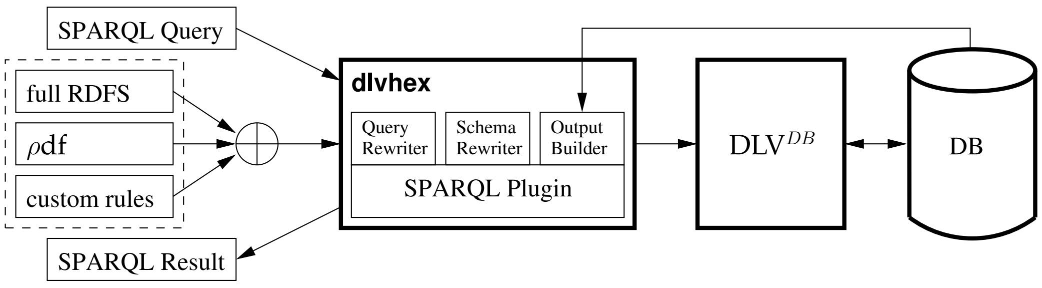

Figure 5.2: The GiaBATAsystem.

The translation from SPARQL to ASP is carried out by the DLVHEX,a solver for the so called ASP^{HEX} programs [99], which features an extensible plugin system allowing for implementing a fully SPARQL compliant semantics, (following the approach proposed in [81]). The ASP Query Engine is implemented by an improved version of the DLV^{DB} system [103] which extends the DLV system with built-in database support and caters for persistent storage of both data and ontology graphs. DLV^{DB} combines the expressive power of DLV with the efficient data management features of DBMSs. Thus, it presents the features of a Deductive Database System (DDS) and can do all the reasoning tasks directly in mass-memory.
The GiaBATAprototype supports the standard SPARQL specifications extended with the syntax outlined above, arbitrary entailment regimes (currently, we support RDFS), and the addition of custom rules. Entailment

rules are expected to be expressed in ASP, and different sets of entailment rules can be attached to the triplestore.

5.1.2 Implementation Issues

From SPARQL to ASP. A SPARQL query Q is transformed into a corresponding Datalog program D_Q using the DLVHEX plugin module, following the approach of [81]. The principle is to break Q down to a series of Datalog rules, whose body is a conjunction of atoms encoding a graph pattern. D_Q is mostly a plain Datalog program in DLVHEX [99] input format, i.e. Datalog with *external predicates* in the DLVHEX language. Without go into details of this transformation (explained along with a full account of the translation in [81, 84]) let us illustrate this step by an example. The following query A asking for persons who are not named "Alice" and optionally their email addresses:

```
select * from <http://alice.org/>
 where { ?X a foaf:Person. ?X foaf:name ?N.
         filter ( ?N != "Alice") optional { ?X foaf:mbox ?M } }
```
(5.1)

is translated to the program D_A as follows:

```
(r1) "triple"(S,P,0,default)     :- &rdf[ "alice.org" ](S,P,0).
(r2) answer1(X_N,X_X,default)    :- "triple"(X_X,"rdf:type","foaf:Person",default),
                                    "triple"(X_X,"foaf:name",X_N,default),
                                    &eval[" ?N != 'Alice' ","N", X_N ](true).
(r3) answer2(X_M,X_X,default) :-  "triple"(X_X,"foaf:mbox",X_M,default).
(r4) answer_b_join_1(X_M,X_N,X_X,default)  :- answer1(X_N,X_X,default),
                                              answer2(X_M,X_X,default).
(r5) answer_b_join_1(null,X_N,X_X,default) :- answer1(X_N,X_X,default),
                                              not answer2_prime(X_X,default).
(r6) answer2_prime(X_X,default) :- answer1(X_N,X_X,default),
                                    answer2(X_M,X_X,default).
(r7) answer(X_M,X_N,X_X)        :- answer_b_join1(X_M,X_N,X_X,default).
```

where the first rule `(r1)` computes the predicate `"triple"` taking values from the built-in predicate *&rdf*. This latter is generally used to import RDF statements from the specified URI.
The following rules `(r2)` and `(r3)` compute the solutions for the filtered basic graph patterns { `?X a foaf:Person. ?X foaf:name ?N.` **`filter`** `(?N != "Alice")` } and { `?X foaf:mbox ?M` }. In particular, note here that the evaluation of **`filter`** expressions is "outsourced" to the built-in predicate *&eval*, which takes a filter expression and an encoding of variable bindings as arguments, and returns the evaluation value (`true`, `false` or `error`, following the SPARQL semantics). In order to emulate SPARQL's **`optional`** patterns a combination of join and set difference operation is

used, which is established by rules `(r4)-(r6)`. Set difference is simulated by using both *null* values and *negation as failure*. According to the semantics of SPARQL, one has to particularly take care of variables which are joined and possibly unbound (i.e., set to the `null` value) in the course of this translation for the general case. Finally, the dedicated predicate *answer* in rule `(r7)` collects the answer substitutions for Q.

From ASP to SQL. For this step we rely on the system DLV^{DB} [103] that implements Answer Set Programming under stable model semantics on top of a DBMS of choice. DLV^{DB} is able to translate ASP programs in a corresponding SQL query plan to be issued to the underlying DBMS. DLV^{DB} does not have, in principle, any practical limitation in the dimension of input data, is capable of exploiting optimization techniques both from the DBMS field (e.g. join ordering techniques [43]) and from the DDS theory (e.g. magic sets [73]), and can easily interact (via ODBC) with external DBMSs. The interested reader can find a complete description of DLV^{DB} and its functionalities in [103]; moreover, the system, along with documentation and examples, are available for download at `http://www.mat.unical.it/terracina/dlvdb`.
RDF Datasets are simply stored in a database D, but the native DLVHEX $\&rdf$ and $\&eval$ predicates in D_Q cannot be processed by DLV^{DB} directly over D. So, D_Q needs to be post-processed before it can be converted into suitable SQL statements. In particular, rule `(r1)` corresponds to loading persistent data into D, instead of loading triples via the $\&rdf$ built-in predicate. In practice, the predicate `"triple"` occurring in program D_A is directly associated to a database table TRIPLE in D. This operation is done off-line by a loader module which populates the TRIPLE table accordingly, while `(r1)` is removed from the program. The $\&eval$ predicate calls are recursively broken down into `WHERE` conditions in SQL statements, as sketched below when we discuss the implementation of **`filter`** statements.
After post-processing, we obtain a program D'_Q, which DLV^{DB} allows to be executed on a DBMS by translating it to corresponding SQL statements. D'_Q is coupled with a mapping file which defines the correspondences between predicate names appearing in D'_Q and corresponding table and view names stored in the DBMS D. For instance, the rule `(r4)` of D_A, results in the following SQL statement issued to the RDBMS by DLV^{DB}:

```
INSERT INTO answer_b_join_1
 SELECT DISTINCT answer2_p2.a1, answer1_p1.a1, answer1_p1.a2, 'default'
 FROM answer1 answer1_p1, answer2 answer2_p2
 WHERE (answer1_p1.a2=answer2_p2.a2)
 AND (answer1_p1.a3='default')
 AND (answer2_p2.a3='default')
 EXCEPT (SELECT * FROM answer_b_join_1)
```

Whenever possible, the predicates for computing intermediate results such as `answer1`, `answer2`, `answer_b_join_1`, ..., are mapped to SQL views rather than materialized tables, enabling dynamic evaluation of predicate contents on the DBMS side.[2]

Schema rewriting. Our system allows for customizing schemes which triples are stored in. It is known and debated [104] that in choosing the data scheme of D several aspects have to be considered, which affect performance and scalability when handling large-scale RDF data. A widely adopted solution is to exploit a single table storing quadruples of form (s, p, o, c) where s, p, o and c are, respectively, the triple subject, predicate, object and context the triple belongs to. This straightforward representation is easily improved [6] by avoiding to store explicitly string values referring to URIs and literals. Instead, such values are replaced with a corresponding hash value.
Other approaches suggest alternative data structures, e.g., property tables [6, 104]. These aim at denormalizing RDF graphs by storing them in a flattened representation, trying to encode triples according to the hidden "schema" of RDF data. Similarly to a traditional relational schema, in this approach D contains a table per each known property name (and often also per class, splitting up the `rdf:type` table).

Our system gives sufficient flexibility in order to program different storage schemes: while on higher levels of abstraction data are accessible via the 4-ary $triple$ predicate, a schema rewriter module is introduced in order to match D'_Q to the current database scheme. This module currently adapts D'_Q by replacing constant IRIs and literals with their corresponding hash value, and introducing further rules which translate answers, converting hash values back to their original string representation. By and large, the chosen database schema for the triples exploits a main table storing quadruples composed of subject, predicate, object, and the source graph of each triple. This representation has been improved [6] by additional relations mapping URIs/literal string to integer values, which avoids string matching in favour of integer comparison.

Magic sets. Notably, DLV^{DB} can post-process D'_Q using the magic sets technique, an optimization method well-known in the database field [13]. The optimized program mD'_Q tailors the data to be queried to an extent significantly smaller than the original D'_Q. The application of magic sets

[2]For instance, recursive predicates require to be associated with permanent tables, while remaining predicates are normally associated to views.

allows, e.g., to apply entailment rules $\mathcal{R}_{RDFS}$ only on triples which might affect the answer to Q, preventing thus the full computation and/or materialization of inferred data.

Implementation of `filter` statements. Evaluation of SPARQL **`filter`** statements is pushed down to the underlying database D by translating filter expressions to appropriate SQL views. This allows to dynamically evaluate filter expressions on the DBMS side.

For instance, given a rule $r \in D_Q$ of the form

$$h(X,Y) \leftarrow b(X,Y), \&eval[f_Y](\mathit{bool}).$$

where the $\&eval$ atom encodes the **`filter`** statement f_Y (representing the filter expression), then r is translated to

$$h(X,Y) \leftarrow b'(X,Y).$$

where b' is a fresh predicate associated via the mapping file to a database view. Such a view defines the SQL code to be used for the computation of f_Y, like

```
CREATE VIEW B' AS ( SELECT X,Y FROM B WHERE F_Y )
```

where `F_Y` is an appropriate translation of the SPARQL **`filter`** expression f_Y at hand to an SQL Boolean condition,[3] while B is the DBMS counterpart table of the predicate b.

5.2 Experiments

Here we present some results of experiments of using our prototype GiaBATA for querying and reasoning over RDFS data. The main goals of our experiments are *(i)* to qualitatively analyse the capabilities of several state-of-the-art triplesore systems, in particular with respect to the problems arising within the scenario depicted in section 4, as well as *(ii)* quantitatively evaluate performances. To this end, we first investigate the persistent storage facilities as well as SPARQL features of some of the state-of-the-art RDF stores, namely, Sesame RDF database 2.3, ARQ 2.6, AllegroGraph 3.2 [4]. Second, in order to illustrate that our approach is practically

[3]A version of this translation can be found in [66].

[4]`http://www.openrdf.org/`, `http://jena.hpl.hp.com/`, and `http://agraph.franz.com/`.

feasible, we present a quantitative performance comparison between our prototype system, GiaBATA, which implements the approach outlined before, and some state-of-the-art triple stores.

5.2.1 Compared Systems

AllegroGraph works as a database and application framework for building Semantic Web applications. The system assures persistent storage and RDFS++ reasoning, a semantic extension including the RDF and RDFS constructs and some OWL constructs. We tested the free Java edition of AllegroGraph 3.2 with its native persistence mechanism.[5]

ARQ is a query engine implementing SPARQL under the Jena framework.[6] It can be deployed on several persistent storage layers, like filesystem or RDBMS, and it includes a rule-based inference engine. Being based on the Jena library, it provides inferencing models and enables (incomplete) OWL reasoning. Also, the system comes with support for custom rules. We used the version 2.6 with RDBMS backend connected to PostgreSQL 8.3.

GiaBATA [58] is our prototype system implementing the SPARQL extensions described above. GiaBATA is based on a combination of the DLV^{DB} [103] and DLVHEX [99] systems, and caters for persistent storage of both data and ontology graphs. The tests were done using development versions of the above systems connected to PostgreSQL 8.3.

Sesame is an open source RDF database with support for querying and reasoning.[7] In addition to its in-memory database engine it can be coupled with relational databases or deployed on top of file systems. Sesame supports RDFS inference and other entailment regimes such as OWL-Horst [102] by coupling with external reasoners. Sesame provides an infrastructure for defining custom inference rules. Our tests have been done using Sesame 2.3 with persistence support given by the native store.

[5]System available at `http://agraph.franz.com/allegrograph/`.

[6]System available at `https://jena.svn.sourceforge.net/svnroot/jena/ARQ/`.

[7]System available at `http://www.openrdf.org/`.

5.2.2 Benchmark Data Set and Queries

For comparison we rely on the Lehigh University Benchmark (LUBM) benchmark suite [49]. The LUBM has been specifically developed to facilitate the evaluation of Semantic Web triplestores in a standard and systematic way. In fact, the benchmark is intended to evaluate the performance of those triplestores with respect to extensional queries over large data sets that commit to a single realistic ontology. It consists of a university domain ontology with customizable and repeatable synthetic data. The Univ-Bench LUBM ontology schema describes (among others) universities, departments, students, professors and relationships among them. Data generation is carried out by the Univ-Bench data generator tool (UBA) whose main generation parameter is the number of universities to consider. This way, we generated several data sets, each containing an increasing number of statements and constructed in such a way that the greater sets strictly contain the smaller ones. Our tests involve the test datasets LUBMn for $n \in \{1, 5, 10, 30\}$, with LUBM30 having roughly four million triples (exact numbers are reported in [49]). The LUBM benchmark provides 14 test queries. Most of these queries basically select subsets of the input data and require, in some cases, inference processes Few of them are intended to verify the presence of certain reasoning capabilities (peculiar of OWL ontologies rather than RDFS) in the tested systems; in fact, they require the management of transitive properties. Queries taken from the LUBM benchmark are $Q1$–$Q7$.

5.2.3 Qualitative Analysis Outcome

First of all, it is worth noting that all systems support persistent storage, either based on RDBMS or on other dedicated backends. As for reasoning expressiveness, all cover RDFS (actually, disregarding axiomatic triples) and partial or non-standard OWL fragments.
Interestingly, although all the systems feature some form of persistence, both reasoning and query evaluation are usually performed in main memory, adopting a materialization strategy. All the systems, except AllegroGraph and ours, adopt a persistent materialization approach for inferring data.
In our tests we focused on querying under ρDF inference regime [74], requiring the capability to match patterns against named graphs. In particular, for qualitative comparison we submit the aforementioned systems to the scenario outlined in section 4.1

Although all systems – along with basic inference – support named graph querying, the investigation revealed that, with the exception of GiaBATA, combining RDFS and named graph queries lead to unexpected (incomplete) behavior as described in section 4.1. In particular, inference is properly handled as long as the query ranges over the whole dataset, whereas RDFS reasoning fails in case of queries using explicit default or named graphs. That makes querying of named graphs involving inference impossible with standard systems.

5.2.4 Quantitative Analysis Outcome

In order to test the additional performance cost of our extensions, we opted for showing how the performance figures change when queries which require RDFS entailment rules (LUBM Q4-Q7) are considered, w.r.t. queries in which rules do not have an impact (LUBM Q1-Q3, see Appendix of [49] for the SPARQL encodings of $Q1$–$Q7$).

First of all, it is worth noting that evaluation times include the data loading times. Indeed, while former performance benchmarks do not take this aspect in account, from the semantic point of view, pre-materialization-at-loading computes inferences needed for complete query answering under the entailment of choice. Moreover, dynamic querying of RDFS moves inference from this materialization to the query step, which would result in an apparent advantage for systems that rely on pre-materialization for RDFS data. Also, the setting of this chapter assumes materialization cannot be performed *una tantum*, since inferred information depends on the entailment regime of choice, and on the dataset at hand, on a *per query* basis. We set a 120min query timeout limit to all test runs.

Our test runs include the following system setup: (i) “Allegro (native)” and “Allegro (ordered)”; (ii) “ARQ”; (iii) “GiaBATA (native)” and “GiaBATA (ordered)”; (iv) “Sesame”. For (i) and (iii), which apply dynamic inference mechanisms, we use “(native)” and “(ordered)” to distinguish between executions of queries in LUBM’s native ordering and in a optimized reordered version, respectively. To appreciate the cost of RDFS reasoning for queries $Q4$–$Q7$, the test runs for (i)–(iv) also include the loading time of the datasets, i.e., the time needed in order to perform RDFS data materialization or to simply store the raw RDF data. The test were done on an Intel P4 3GHz machine with 1.5GB RAM under Linux 2.6.24. Experiments are enough for comparing performance trends, so we didn’t consider at this stage larger instances of LUBM. The detailed outcome of the test results are summarized in Fig. 5.3. For the RDF test queries $Q1$–$Q3$, GiaBATA is

able to compete for $Q1$ and $Q3$. The systems ARQ and Sesame turned out to be competitive for $Q2$ by having the best query response times, while Allegro (native) scored worst. For queries involving inference ($Q4$–$Q7$) Allegro shows better results. Interestingly, systems applying dynamic inference, namely Allegro and GiaBATA, query pattern reordering plays a crucial role in preserving performance and in assuring scalability; without reordering the queries simply timeout. In particular, Allegro is well-suited for queries ranging over several properties of a single class, whereas if the number of classes and properties increases ($Q7$), GiaBATA exhibits better scalability. Finally, a further distinction between systems relying on DBMS support and systems using native structures is disregarded, and since figures (in logarithmic scale) depict overall loading and querying time, this penalizes in specific cases those systems that use a DBMS.
Our initial experiments have shown that although dynamic querying does more computation at query-time, it is still competitive for use cases that need on-the-fly construction of datasets and entailment regimes. Especially here, query optimization techniques play a crucial role, and our results suggest to focus further research in this direction.

Remark. In this section we reported some up-to-date experimental results being related to our recent investigations. Previous tests including a more richer and comprehensive analysis can be found in [59]. These results were carried out adopting the same LUBM benchmark suite (but with respect to bigger datasets and over a wider range of queries). Moreover these previous experiments have been successfully applied on a real-world dataset, namely, the DBLP database containing a large number of bibliographic descriptions on major computer science journals and proceedings. In particular, for comparing both scalability and expressiveness, we exploited queries of increasing complexity, ranging from simple selections to queries requiring different forms of inferences over the data. Notably, we tried additional queries, aiming at testing higher features like the capability to aggregate data and to perform a transitive closure over the underlying data and, consequently requiring the capability to express recursion. The interested reader can find them in the Appendix to [59] [8]. Results clearly show how the approach successfully works for all the query in terms of performance and expressivity, being capable to cover the whole set of queries and being competitive with respect to others systems.

[8] `http://www.mat.unical.it/terracina/dlvdb/JLC/Appendix.pdf`.

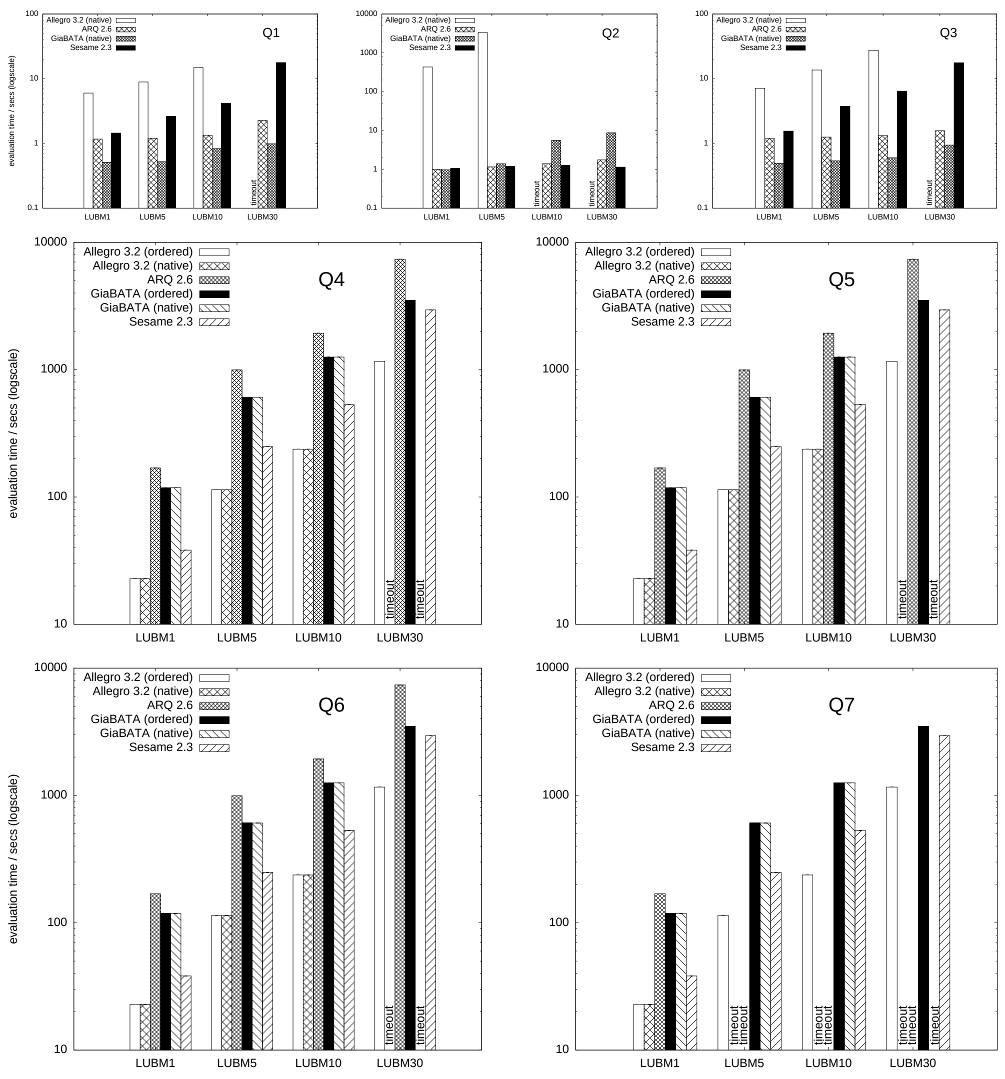

Figure 5.3: Evaluation

Versatile semantic Modeling of Knowledge using ASP

Beside the perspective to exploit ASP as an implementation platform for the Semantic Web, a question arise whether it can be also explored as a suitable basis for ontology languages which form the backbone of the Semantic Web. Indeed, the Semantic Web offers many languages for ontologies (RDFS, OWL, DL-Lite, Frame Logic), each having his own model theoretical semantics. That encourages and motivates investigations about relationships between these languages with other KR&R formalisms, devising alternative solutions for modeling semantics of Knowledge Representation Language of interest. In most cases, how to model these semantics to ASP has been shown, in other cases, reductions are not clearly known.

Among them, we focus on Frame Logic which has been proposed as representation language for the SW ([61]) and whose relevancy is witnessed by recent projects such as WSMO [29, 91]. Moreover, its features play a crucial role in the ongoing activity of the RIF Working group [19, 17] [9]. Frame Logic was originally defined as extension of first-order logic [62], allowing for an object-oriented (frame-based) style of modeling, specifying methods, as well as generalization/specialization and instantiation relationships. Later on, a well-founded semantics, satisfactorily dealing with non-monotonic inheritance has been introduced [107]. Beside this setting, the original paper ([62]) already defined a Logic Programming-style semantics for the subset of Frame Logic based on Horn logic and there exist several implementations of Frame Logic programming.
Its object-oriented features and its non-monotonic variant, capable to deal with typical non-monotonic features such as object oriented inheritance make the Frame Logic language both an important methodology and a tool for modeling ontologies in the context of Semantic Web. Especially, the non-monotonic semantics is often requested by Semantic-Web design-

[9]http://www.w3.org/2005/rules/wiki/RIF_Working_Group..

ers in cases where the reasoning capabilities of the Ontology layer turn out to be too limiting, since they are based on monotonic logics.

In this respect, the Answer Set Programming paradigm has some attractive feature which make interesting to consider the possibility of defining a frame-based language under this setting.
Indeed, ASP allows to model declaratively nondeterminism and, in particular, it shares with Frame Logic under well-founded semantics the possibility to reason about ontologies using non-monotonic constructs, included non-monotonic inheritance, as it is done in some ASP extensions conceived for modeling ontologies [90]. Nonetheless, ASP misses the useful Frame Logic syntax and higher order reasoning capabilities, that are widely acknowledged as useful for various tasks and are essential in the context of meta-reasoning. Motivated by this fact and considering that to date most attention around Frame Logic is around Frame Logic Programming, we investigated about possible usage of the logic programming paradigm of ASP in this respect. Therefore, in the following we will restrict ourselves to the Logic Programming semantics for Frame Logic and propose an extension of ASP which aims at effectively integrating the flexible and intuitive way of representing knowledge in ASP with some of the Frame Logic features.
Based on an elemental model theoretic semantics (ASP) on which to plugin axiomatic modules for modelling/customizing specific semantical behaviors, our approach tours out to be flexible enough to be easily extended for other languages. As such it provides a "testbed" for modeling a variety of semantics, with possibility to play and practice with multiple ones.

Chapter 6

Frame Logic under ASP semantics

Here we introduce a novel framework for coping with frame-like syntax and higher-order reasoning within an Answer Set Programming environment, introducing the family of Frame Answer Set Programs (FAS programs) (FAS). First, we present the syntax of the language includes the possibility to manipulate nested molecules, class hierarchies, basic method signatures and contexts (called *framespaces*). We illustrate in which terms contexts can be exploited for manipulating hybrid knowledge bases having many data sources working under different entailment regime. We provide the model-theoretic semantics of FAS programs in terms of their *answer sets*. Then, by means of a translation to an ordinary answer set program, we show the corresponding operational semantics of FAS programs. Notably, the proposed language is purposely designed so that *inheritance* behavior and other features of the language can be easily customized by the introduction of specialized axiomatic modules (section 6.2), which can be modeled on purpose by advanced developers of ontology languages. We describe how to use the language for modeling and axiomatizing knowledge, and proves some properties of the axiomatic modules presented (section 6.3). 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 Frame Logic under first order semantics. Finally, we present the system (DLT), able to deal with programs in Frame Logic like syntax, and featuring higher-order reasoning.

6.1 The Framework of the FAS Programs

Frame Logic [62, 107] is a knowledge representation and ontology 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.
The basic idea behind Frame Logic is to consider complex data types as in object-oriented databases, combine them with logic and use the result as a programming language. Frame Logic eases the burden of declaring properties of instance data. With respect to the flat relational model (and to its logic counterpart, where tables are modeled as predicates), it is possible to "talk" about all the facts related to a single individual within a single structure, i.e. the *molecule*. Moreover, it eases the burden of reasoning about classes: higher order reasoning is native, so that the border between the notion of class and individual is smooth. It is then possible to reason about classes with the same ease of use as classic logic programming offers when it is necessary to reason on individuals.

6.1.1 Syntax of FAS Programs

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 Answer Set Programming programs, 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 Frame Logic [62]. 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:

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

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 $brown : employee@IBM$. A Frame Space directive tells how frames are mapped to regular atoms, and can be used for defining modules where each predicate has local scope within a given frame space. 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*). Informally, a *frame molecule* asserts that the object has some properties as specified by the attribute expressions listed inside the brackets. 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 $p \rightharpoonup\!\!\!\rightharpoonup \{v_1, \ldots, v_n\}$, where p (*the property/attribute name*) is a term, and $v_1, \ldots, v_n$ (*the attribute values*) are either terms or frame molecules. An attribute expression defines an association between an *attribute name* and one or multiple *values* than it can take. A *negative attribute expression* is a negated positive attribute expression. An *attribute expression* is either a positive attribute expression or a negative attribute expression. Here and in the following, the meta-symbols $\rightharpoonup$ and $\rightharpoonup\!\!\!\rightharpoonup$ 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 $s[p \rightharpoonup\!\!\!\rightharpoonup \{v\}]@f$, 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 $B(r)$.

Remark. Every object name refers to *exactly one* object, although molecules starting with the same *subject* may be combined. Since the value referred to an object attribute can be frames, molecules can be *nested*.

Example 6.1.1 *The following is a frame molecule:*

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

This defines membership of the subject *brown* to the *employee* class and asserts some values corresponding to the properties bind to this object. This frame molecule says *brown* is *male* (as expressed by the value of the attribute *gender*), is *married* to another employee identified by the subject *pink*. *brown* knows *java* and *asp* languages, as suggests the values of the *skill* property, while he has a *salary* equal to *800*. We may define a new frame molecule, like this, collecting information about the employee encoded by the subject *pink*, but we can also combine this information nesting frame molecules, as follows:

$$\begin{array}{lll} brown : employee[& surname \rightarrow \text{“}Mr.\,Brown\text{”}, & \\ & skill \twoheadrightarrow \{java, asp\}, & \\ & salary \rightarrow 800, & \\ & gender \rightarrow male, & \\ & married \rightarrow pink : employee[& surname \rightarrow \text{“}Mrs.\,Pink\text{”}, \\ & & skill \twoheadrightarrow \{html, asp\}, \\ & & salary \rightarrow 900, \\ & & gender \rightarrow female, \\ & & married \rightarrow brown\,] \\ ]. & & \end{array}$$

Example 6.1.2 *The following is an example of logic rule defining the profile a particular employee must have in order to be selected for project* $p3$*. We encoded this rule using* strong *and* naf *nested negation.*

$$\begin{array}{ll} E[inProject \twoheadrightarrow p3] \vee E[\neg inProject \twoheadrightarrow p3] \leftarrow & X : employee, \\ & E : employee[skill \twoheadrightarrow \{c,\, perl\}, \\ & not\ married \rightarrow X : employee[\\ & not\ skill \twoheadrightarrow \{c,\, perl\}\,]]. \end{array}$$

This means that candidates to the project team *p3* are employees knowing *c* and *perl* programming languages, but not married to another employee not knowing the same programming languages.

Example 6.1.3 *The following one rule program is a valid* FAS *program. Intuitively, it represents the fact that each person is* male *or* female.

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

6.1.2 Semantics of FAS Programs

Our approach is set in between a pure model theoretic semantics (proper of Frame Logic and many of its extensions [62, 107]), and a pure "rewriting" semantics, in which inheritance is specified by means of an ad-hoc translation to logic programming [60]. 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.
In the following, we provide the model-theoretic semantics of FAS programs in terms of their *answer sets*. Then, we provide, by means of a translation to an ordinary answer set program, the corresponding operational semantics of FAS programs.
Notably, the basic stable model semantics for FAS programs does not purposely fix a special meaning for the traditional operators of Frame 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.

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 [47], 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 $brown[color \rightarrow yellow]$).

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:

- $(E5)$ $I \models s[p \rightharpoonup\!\!\!\rightharpoonup \{m_1, \ldots m_n\}]@f$ iff $I \models s[p \rightharpoonup\!\!\!\rightharpoonup \{m_i\}]@f$, 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 $I \models s \diamond c\,@f \wedge s[\overline{v}]@f$;
- $(E8)$ $I \models s[p \rightharpoonup s'[\overline{v}]]@f$ iff $I \models s[p \rightharpoonup s']@f \wedge s'[\overline{v}]@f$;
- $(E9)$ $I \models s[p \rightharpoonup\!\!\!\rightharpoonup \{s'[\overline{v}]\}]@f$ iff $I \models s[p \rightharpoonup\!\!\!\rightharpoonup \{s'\}]@f \wedge s'[\overline{v}]@f$;
- $(E10)$ $I \models s[p \rightharpoonup s'[\overline{v}]@f']@f$ iff $I \models s[p \rightharpoonup s']@f \wedge s'[\overline{v}]@f'$;
- $(E11)$ $I \models s[p \rightharpoonup\!\!\!\rightharpoonup \{s'[\overline{v}]@f'\}]@f$ iff $I \models s[p \rightharpoonup\!\!\!\rightharpoonup \{s'\}]@f \wedge s'[\overline{v}]@f'$.

[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.

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 $\mathrm{MM}(P)$.

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 $\{a_1 \vee \cdots \vee a_n \leftarrow b_1, \cdots, b_k, \text{not } b_{k+1}, \cdots, \text{not } b_m\}$ is in $grnd(P)$ and $I \models \text{not}\, b_{k+1} \wedge \ldots \wedge \text{ not } b_m$. 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) *[86, 47]*. 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 Frame 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 $\approx\!> \in \{\Rightarrow, \Rrightarrow\}$;
- $(F5)$ if $c[m \Rightarrow v]$, $a\!:\!c$ and $a[m \rightarrow w] \in M_f$ then $w\!:\!v \in M_f$;
- $(F6)$ if $c[m \Rrightarrow v]$, $a\!:\!c$ 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 6.1.4 *The program in Example 6.1.3 together with the fact* $brown\!:\!person$. *has two answer sets,* $M_1 = \{brown\!:\!person,\ brown[\,gender \rightarrow \text{“}male\text{”}\,]\}$ *and* $M_2 = \{brown\!:\!person,\ brown[gender \rightarrow \text{“}female\text{”}]\}$. *Both* M_1 *and* M_2 *are F-models. Note that* $M_3 = \{\ brown : person,\ brown[\ gender \rightarrow \text{“}female\text{”}\ ],\ brown[\,gender \rightarrow \text{“}male\text{”}\,]\}$ *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* $\leftarrow brown[gender \rightarrow \text{“}male\text{”}]$ *and* $\leftarrow brown[gender \rightarrow \text{“}female\text{”}]$ *has no answer set* [2].

6.1.2.1 From FAS programs to higher order programs

We provide here the semantics of FAS programs in terms of a translation to a (plain) higher-order ASP program. We show how to reduce the Frame Logic like formalism embedded in our hybrid framework to ASP, thus allowing to manipulate frames with logic programming techniques. This operational semantics is defined through a suitable algorithm which is able, given a FAS programs containing frame structures, to produce an equivalent plain higher order ASP program.
Roughly speaking, the idea is to introduce new predicate names wrapping properties and classes. Classes are mapped to unary predicates, while properties are mapped to unary or binary predicates. Then, a FAS program is *unfolded* in order to replace frame atoms with their equivalent predicates.
The algorithm providing the semantics is called *Standardize Algorithm* ($\mathcal{S}$): it takes as input a *FAS program* P containing frame atoms; the output is a plain higher order program R. The Answer Sets of P are defined as to be the answer sets of R. The algorithm is sketched in Figure 6.1. In order to better explain how S works, we show how a frame structure is examined and processed. For instance, if we consider the following frame:

$$\begin{array}{ll} E[inProject \twoheadrightarrow p3] \vee E[-inProject \twoheadrightarrow p3] \leftarrow & X : employee, \\ & E : employee[\\ & skill \twoheadrightarrow \{c++, perl\}, \\ & not\ married \rightarrow X\,]. \end{array}$$

The application of S generates this output:

$$\begin{array}{rcl} inProject(E,\, p3) \vee -inProject(E,\, p3) & \leftarrow & employee(X), skill(E,\, c++), \\ & & skill(E,\, perl), employee(E), \\ & & not\, aux\_e(E,\, X). \\ aux\_e(E,\, X) & \leftarrow & married(E,\, X). \end{array}$$

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

6.2 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.

$$\begin{array}{ll} c_1: & A::B \leftarrow A::C,\ C::B. \\ c_2: & A::A \leftarrow X:A. \\ c_3: & \leftarrow A::C,\ C::A,\ A \neq C. \\ c_4: & X:C \leftarrow X:D,\ D::C. \end{array}$$

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 Frame 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:

$$\begin{array}{ll} f_5: & \leftarrow A[M \rightarrow V], A[M \rightarrow W], V \neq W \\ f_6: & \leftarrow A[M \bullet\!\!\rightarrow V], A[M \bullet\!\!\rightarrow W], V \neq W \end{array}$$

Structural and behavioral inheritance. We show here how to model some peculiar types of inheritance, such as structural and behavioral inheritance. Structural inheritance is usually associated to the operator $\Rightarrow$. Let P_1 be the following example program:

$$\begin{array}{l} webDesigner::javaProgrammer.\ javaProgrammer::programmer. \\ webDesigner::htmlProgrammer.\ javaProgrammer[salary \Rightarrow medium]. \\ htmlProgrammer[salary \Rightarrow low]. \end{array}$$

For short, we denote in the following *webDesigner* as *wd*, *javaProgrammer* as *jp* and *htmlProgrammer* as *hp*. Under structural inheritance, as defined in [62], prop-

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

erty 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 $\Rightarrow\!\!\!\!\Rightarrow$.

$$\begin{array}{ll} s_7 : & D[A \Rightarrow T] \leftarrow D{::}C,\ C[A \Rightarrow T]. \\ s_8 : & D[A \Rightarrow\!\!\!\!\Rightarrow T] \leftarrow D{::}C,\ C[A \Rightarrow\!\!\!\!\Rightarrow T]. \end{array}$$

Note that s_5 (resp. s_6) do not enforce any relationship between "$\Rightarrow$" and "$\rightarrow$" (resp. "$\Rightarrow\!\!\!\!\Rightarrow$" and "$\twoheadrightarrow$") as in [62]. We will discuss this issue later in the section.

Behavioral inheritance [107], allows instead non-monotonic 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 [107][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 [107] 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 $P_1 \models_{\cap \mathcal{B} \cup \mathcal{F} \cup \mathcal{C}} wd[income \bullet\!\!\rightarrow 1200]$ hold. The above behavior can be enforced by defining $\mathcal{B}$ as follows

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

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

[4]Note that in [107] the above semantics is conventionally associated to the $\rightarrow$ operator, while we will use $\bullet\!\!\rightarrow$.

the possible answer sets. Thus, similarly to "well-founded optimism" semantics, we obtain that $P_1 \not\models_{\mathcal{C}\cup\mathcal{B}} wp[income \bullet\!\!\to X]$ 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:

$programmer[salary \Rightarrow integer].$
$g : programmer[salary \rightarrow aSalary].$
$\leftarrow X : programmer[salary \rightarrow Y], \text{not}\, Y : integer$[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[105]. 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 Frame 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).

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

Whenever required, FAS programs can be coupled with axiomatic modules encoding both well-typed and constructive axioms. The following axiomatic module $\mathcal{CO}$ encodes constructively how the operators $\Rightarrow$ and $\rightarrow$ can be related each other:

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

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

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

6.3 Properties of FAS programs

FAS programs have some property of interest. First, Frame 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 6.3.1 *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 Frame Logic . It is matter of future research to investigate on the relationship between FAS programs and Frame 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 [74]), 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 $a@c$.

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 3 *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* [92] 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 6.3.2 *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 $P \cup A@a \cup A@b^M$. □

6.4 The DLT System

FAS programs have been implemented within the DLT environment [25] which extends the DLV system with *template predicates*, *frame logic* constructs and *higher order* predicate names. Anyway, the proposed paradigm does not rely on DLV special features, and it is easily generalizable.

6.4.1 The System Architecture and Work-flow

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 [64], DLVHEX [99] and the recent DLV-complex[6]. 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.

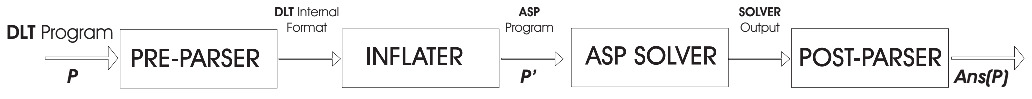

Figure 6.2: Architecture of DLT System.

[6]http://www.mat.unical.it/dlv-complex.

The overall architecture of the system is shown in Figure 6.2. Roughly, the DLT system works as follows. A FAS program P is sent to a DLT*pre-parser*, which performs *syntactic checks*, converts frame syntax to plain syntax (by applying the algorithm $\mathcal{S}$ defined in Section 6.1.2.1), and builds an internal representation of P. The DLT *Inflater* produces an equivalent program P' by processing and eliminating other special constructs of the language, such as templates; P' is piped towards an answer set solver. The answer sets $ans(P')$ of P', computed by the solver are then converted in a readable format through the *Post-parser* module, which filters out from $ans(P')$ information about predicates and rules that were internally generated.
DLT allows the syntax presented in this thesis 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.
When the directive "@." is used, the systems switch's to the default frame space, thus triggering the traditional behavior of the system.

6.4.2 Extending the ASP language using DLT

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 [65], allowing in many case to represent information in a more succinct way.

Example 6.4.1 *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.

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

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 [25].

Solvers support. DLT can virtually support any solver that accepts inputs in the format generated by DLT. Models produced by the external solver are then parsed back to the DLT syntax. The `-solver=[pathname]` option allows to specify the path of the solver. Compatibility with the systems DLV [64], DLV-EX [24], DLVHEX [39] is provided; compatibility with S-models is guaranteed within a subset of the language. A detailed compatibility table is available on the DLT web site.

Function Symbols. Besides constant and variable terms, the DLT parser allows also functional terms. Solvers allowing function symbols are thus ready to be coupled with DLT.

```
**Standardize** (**INPUT**: $P$ containing frames  **OUTPUT**: $R$ without frames)
Let $R = P$;
while $R$ contains frame literals do
   let $r \in R$ be a rule containing frame atoms;
   while $r$ contains frame literals do
      remove frame $f$ from $r$;
      let $o$ be the subject, $L$ the set of attributes, and $\mathbf{X}$ the set of variables of $f$;
      case $f$ appeared in the body of $r$:
         if $f$ is positive then
            if $f$ has class $c$ then
               add $c(o)$ to the body of $r$;
            for each attribute expression $e \in L$ do
               let $a$ be the name, and $V$ the set of values of $e$;
               if $e$ is positive then
                  if $V$ is empty then
                     add $a(o)$ to the body of $r$;
                  else
                     for each term $t \in V$ do
                        add $a(o,t)$ to the body of $r$;
                     for each molecule $m \in V$ with subject $s$ do
                        add $a(o,s)$ and $m$ to the body of $r$;
               else
                  let $e$ be in the form $not\ e'$;
                  add the frame $not\ o[e']$ to the body of $r$;
         else (Let $f$ in form $not\ f'$)
            add to $r$ a new fresh literal not $aux_f(\mathbf{X})$;
            add to $R$ a new rule $aux_f(\mathbf{X}) \leftarrow f'$;
      case $f$ appeared in the head of $r$:
         if $f$ is in the form $o : c$ (resp. in the form $o[a \rightarrow v]$) then
            add to the head of $r$ the literal $c(o)$ (resp. $a(o,v)$);
         else
            add to the head of $r$ a new atom $aux_f(\mathbf{X})$;
            if $f$ has class $c$ then
               add $c(o) \leftarrow aux_f(\mathbf{X})$ to $R$;
            for each attribute expression $e \in L$ do
               let $a$ be the name, and $V$ the set of values of $e$;
               if $V$ is empty then
                  add $a(o) \leftarrow aux_f(\mathbf{X})$ to $R$;
               else
                  for each term $t \in V$ do
                     add $a(o,v) \leftarrow aux_f(\mathbf{X})$ to $R$;
                  for each molecule $m \in V$ with subject $s$) do
                     add $a(o,s) \leftarrow aux_f(\mathbf{X})$ and $m \leftarrow aux_f(\mathbf{X})$ to $R$;
```

Figure 6.1: The Standardize Algorithm.

Related work, future developments and Conclusions

In this work we presented a formal approach aimed at exploiting the Answer Set Programming formalism for tackling several issues coming from traditional and upcoming scenarios of the the Semantic Web. We also focused on applicative possibilities of our approach proposing an implementation addressing several problems, thus suggesting possible improvement to the state-of-the-art systems.

ASP for Semantic Web Application We have first formally shown how RDF(S) can be mapped without loss of semantics to an Answer Set Programming language, providing a complete translation of all the normative RDF(S) semantics into declarative logic programming. Afterward, we presented a framework for dynamic querying of RDFS data, which extends SPARQL by two language constructs: **`using ontology`** and **`using ruleset`**. The former construct is geared towards dynamically creating the dataset, whereas the latter adapts the entailment regime of the query. We have shown that our extension conservatively extends the standard SPARQL language and that by selecting appropriate rules in **`using ruleset`**, we may choose varying rule-based entailment regimes at query-time.
We completed our investigation by presenting an efficient and reliable ASP-based architecture for querying RDF ontologies and we have experimentally proven that our solution is competitive with respect to several state-of-the-art systems. Our initial experiments have shown that although dynamic querying does more computation at query-time, it is still competitive for use cases that need on-the-fly construction of datasets and entailment regimes.
Compared with off-the-shelf RDF stores and SPARQL engines, the system aims at showing how it is possible to offer more flexible support for rule-based RDFS and (fragments of) OWL entailment regimes by enabling custom reasoning via rules, as well as at giving the possibility to choose the reference ontology on a *per query* basis; this allows for dynamically changing scenarios in the Semantic Web to be safely taken into account.

The work is similar in spirit to [32], where it is shown how RDF, RDFS and ERDFS can be ported to Frame Logic (and, to a large extent, to Datalog), and thus implemented in a standard reasoner. The above work does not address explicitly the problem of treating the infinite set of axiomatic triples of RDFS in a finite context. Also, in [74] a deductive calculus is presented which is sound and complete for the ρDF fragment of the language, and paves the way to its implementation in a deductive system.
It is worth pointing out also that important efforts in the Semantic Web community aim at integrating Ontologies with Rules under the stable model semantics (e.g. [38, 39, 72]). In this context, the abovementioned works highlight that the possibility of exploiting a Datalog-like language to express (or integrate) ontologies with a query/rule language provides important benefits. A work explicitly addressing RDF and proposing the idea of extending RDF graphs with negation and stable models is [8].
There are earlier approaches to integrate ontological inference in a logic programming environment. As opposed to SWI Prolog's Semantic Web library [106] which also offers SPARQL support, we support Answer Set Programming as underlying LP paradigm, instead of Prolog. In contrast to OntoDLV [89], which supports proprietary ontology and query languages, the focus of GiaBATA is on compliance with Web standards such as SPARQL, RDF(S), OWL, and RIF.

As a matter of future work, we want to point out that the fully Answer Set Programming based architecture, as relying on logic programming techniques and SQL, provides entry points for several well-known logic-level optimization and deductive database optimization techniques. Applying optimization techniques from both the database and logic programming area should boost evaluation performance. For instance, one could pursue an integrated approach to query optimization based on both statistical and declarative/logical optimizations. This includes developing novel techniques for optimization tasks, such as magic sets and join reordering techniques ([10-11]). We currently work on the efficiency of **`filter`** expressions by rewriting them to SQL queries over the underlying database, see [6, 66, 104]. Moreover, one of the next steps includes the support of arbitrary database schemes. Extensions of SPARQL by aggregates and custom built-ins presented in earlier works [84] carry over to our persistent storage version with minor modifications. In particular, we plan to integrate the system with extensions of SPARQL enhancing expressiveness and practical capabilities, introducing novel features natively available within ASP systems (aggregates, custom built-ins, recursive constructs, soft and weak constraints, interfacing with external reasoners). We are going to add support for extended graphs, that is SPARQL views as defined also in [84].
Furthermore, we aim at conducting a proper computational analysis as it has been done for Hypothetical Datalog [20], in which truth of atoms is conditioned by hypothetical additions to the dataset at hand. Likewise, our framework allows to add ontological knowledge and rules to datasets before querying: note how-

ever that, in the spirit of [50], our framework allows for hypotheses (also called "premises") on a per query basis rather than a per atom basis.

ASP for modeling semantics We presented a framework that allows to enrich an Answer Set Programming language with frame-like syntax and higher order reasoning.

FAS programs have some peculiar differences with respect to the original Frame Logic . Importantly, while well-founded semantics [44] is at the basis of the non-monotonic semantics of Frame 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 Frame 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 [44].

FAS answer sets should not be confused with the notion of *stable object model* given in [107].

Since Frame 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 Frame Logic systems Flora and Florid ([108, 67]) share with FAS programs the ability to work both on the level of concepts and attributes and on instances.

Several Semantic Web initiatives point to Frame Logic as rule-based language core, like SWSL ([4]) and WSML ([33]) which in its more powerful variants is based on Frame Logic layered on top of Description Logic [31].

Frame Logic has been investigated as a logical way to provide reasoning capability on top of RDF in the system TRIPLE ([94]) 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 ([63]).

Several works share some point in common with this thesis in the field of Answer Set Programming. An inspiring first definition of Frame Logic under stable model semantics can be found in [31]. The fragment considered focuses on first order Frame 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 [39]. Contexts were investigated under stable model semantics also in [82]. 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 [21] 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 [65], although we do not allow the use of negation in the head of rules. A different approach to non-monotonic inheritance in the context of stable model semantics was proposed in [23], 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 [90] and [89].

As a matter of future research, we plan to investigate thoroughly about the relationship between Frame Logic under well-founded semantics and similar formalizations of non-monotonic inheritance under stable model semantics. Also, the usage of arbitrarily nested molecules, including negation as failure, deserve further investigation.

Bibliography

[1] Allegrograph. `http://agraph.franz.com/allegrograph/`.

[2] Arq. `https://jena.svn.sourceforge.net/svnroot/jena/ARQ/`.

[3] Schemaweb. `http://www.schemaweb.info/`.

[4] Semantic Web Services Language (SWSL). `http://www.w3.org/Submission/SWSF-SWSL/`.

[5] Sesame. `http://www.openrdf.org/`.

[6] D. J. Abadi, A. Marcus, S. Madden, and K. J. Hollenbach. Scalable Semantic Web Data Management Using Vertical Partitioning. In *Proceedings of the 33rd International Conference on Very Large Data Bases (VLDB 2007), University of Vienna, Austria, September 23-27, 2007*, pages 411–422. ACM, 2007.

[7] S. Abiteboul. Querying semi-structured data. In F. N. Afrati and P. G. Kolaitis, editors, *Database Theory - ICDT '97, 6th International Conference, Delphi, Greece, January 8-10, 1997, Proceedings*, volume 1186 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 1997.

[8] A. Analyti, G. Antoniou, C. V. Damsio, and G. Wagner. Stable Model Theory for Extended RDF Ontologies. In *4th International Semantic Web Conference, ISWC 2005, Galway, Ireland*, pages 21–36, Nov. 2005.

[9] R. Angles and C. Gutierrez. The Expressive Power of SPARQL. In *Proceedings of the 7th International Semantic Web Conference (ISWC 2008)*, LNCS, pages 114–129. Springer, 2008.

[10] S. Auer, C. Bizer, G. Kobilarov, J. Lehmann, R. Cyganiak, and Z. Ives. DBpedia: A Nucleus for a Web of Open Data. In *Proceedings of 6th International Semantic Web Conference, 2nd Asian Semantic Web Conference (ISWC+ASWC 2007)*, pages 722–735. November 2008.

[11] F. Baader. *The Description Logic Handbook: Theory, Implementation, and Applications.* Cambridge University Press, September 2007.

[12] B. P.-S. B.Cuenca Grau, I.Horrocks and U.Sattler. Next Steps for OWL. *In Proc. of the Second OWL Experiences and Directions Workshop*, 216, 2006.

[13] C. Beeri and R. Ramakrishnan. On the power of magic. *The Journal of Logic Programming*, 10(3-4):255–299, 1991.

[14] T. Berners-Lee and M. Fischetti. *Weaving the Web; The Original Design and Ultimate Destiny of the World Wide Web, by Its Inventor*. Harper Collins, 2000.

[15] T. Berners-Lee, J. Hendler, and O. Lassila. The Semantic Web. May 2001.

[16] D. Berrueta, D. Brickley, S. Decker, S. Fernndez, C. Grn, A. Harth, T. Heath, K. Idehen, K. Kjernsmo, A. Miles, A. Passant, A. Polleres, L. Polo, and M. Sintek. SIOC Core Ontology Specification. W3c member submission, W3C, June 2007.

[17] H. Boley and M. Kifer. RIF Core Design. *W3C Editor's Draft*, 2007.

[18] H. Boley and M. Kifer. RIF Basic Logic Dialect, July 2009. W3C Working Draft, available at `http://www.w3.org/TR/2009/WD-rif-bld-20090703/`.

[19] H. Boley, M. Kifer, P.-L. Pa̱trânjan, and A. Polleres. Rule Interchange on the Web. In *Reasoning Web 2007*, volume 4636, pages 269–309. Springer, Sept. 2007.

[20] A. J. Bonner. Hypothetical datalog: Complexity and expressibility. *Theoretical Computer Science*, 76(1):3–51, 1990.

[21] G. Brewka and T. Eiter. Equilibria in heterogeneous nonmonotonic multi-context systems. In *AAAI*, pages 385–390, 2007.

[22] D. Brickley and L. Miller. The Friend Of A Friend (FOAF) vocabulary specification, November 2007.

[23] F. Buccafurri, W. Faber, and N. Leone. Disjunctive Logic Programs with Inheritance. *TPLP*, 2(3), May 2002.

[24] F. Calimeri, S. Cozza, and G. Ianni. External sources of knowledge and value invention in logic programming. *AMAI*, 50(3–4):333–361, 2007.

[25] F. Calimeri and G. Ianni. Template programs for disjunctive logic programming: An operational semantics. *AI Communications*, 19(3):193–206, 2006.

[26] A. Colmerauer and P. Roussel. The birth of Prolog. In T. J. Bergin and R. G. Gibson, editors, *History of Programming Languages II*, pages 331–367. ACM, New York, NY, USA, 1996.

[27] R. D.Brickley and B. (eds.). RDF Vocabulary Description Language 1.0: RDF Schema. Technical report, W3C, February 2004. W3C Recommendation.

[28] J. de Bruijn. *Semantic Web Language Layering with Ontologies, Rules, and Meta-Modeling*. PhD thesis, University of Innsbruck, 2008.

[29] J. de Bruijn, C. Bussler, J. Domingue, D. Fensel, M. Hepp, M. Kifer, B. KÃ*P*nig-Ries, J. Kopecky, R. Lara, E. Oren, A. Polleres, J. Scicluna, and M. Stollberg. WSMO Final Draft, 2005. `http://www.wsmo.org/TR/d2/v1.2/`.

[30] J. de Bruijn, E. Franconi, and S. Tessaris. Logical reconstruction of normative RDF. In *OWL: Experiences and Directions Workshop (OWLED-2005)*, Galway, Ireland, 2005.

[31] J. de Bruijn and S. Heymans. Translating ontologies from predicate-based to frame-based languages. In *RuleML*, pages 7–16, 2006.

[32] J. de Bruijn and S. Heymans. Logical Foundations of (e)RDF(S): Complexity and reasoning. pages 86–99. 2008.

[33] J. de Bruijn, H. Lausen, A. Polleres, and D. Fensel. The Web Service Modeling Language WSML: An Overview. In *ESWC*, pages 590–604, 2006.

[34] J. Dix. Semantics of logic programs: Their intuitions and formal properties. an overview. In *Logic, Action and Information. Proceedings of the Konstanz Colloquium in Logic and Information (LogIn'92)*, pages 241–329. DeGruyter, 1995.

[35] T. Eiter, W. Faber, N. Leone, and G. Pfeifer. Declarative Problem-Solving Using the DLV System. In J. Minker, editor, *Logic-Based Artificial Intelligence*, pages 79–103. Kluwer Academic Publishers, 2000.

[36] T. Eiter, G. Gottlob, and H. Mannila. Disjunctive Datalog. *TODS*, 22(3):364–418, sep 1997.

[37] T. Eiter, G. Ianni, T. Krennwallner, and A. Polleres. Rules and Ontologies for the Semantic Web. In *Reasoning Web*, pages 1–53, 2008.

[38] T. Eiter, G. Ianni, T. Lukasiewicz, R. Schindlauer, and H. Tompits. Combining Answer Set Programming with Description Logics for the Semantic Web. *Artificial Intelligence*, 172(12-13):1495–1539, 2008.

[39] T. Eiter, G. Ianni, R. Schindlauer, and H. Tompits. A Uniform Integration of Higher-Order Reasoning and External Evaluations in Answer Set Programming. In *International Joint Conference on Artificial Intelligence (IJCAI) 2005*, pages 90–96, Edinburgh, UK, Aug. 2005.

[40] T. Eiter, N. Leone, C. Mateis, G. Pfeifer, and F. Scarcello. The kr system `dlv`: Progress report, comparisons and benchmarks. In A. G. Cohn, L. K. Schubert, and S. C. Shapiro, editors, *Proceedings Sixth International Conference on Principles of Knowledge Representation and Reasoning (KR'98)*, pages 406–417. Morgan Kaufmann Publishers, 1998.

[41] W. Faber, N. Leone, and G. Pfeifer. Recursive aggregates in disjunctive logic programs: Semantics and complexity. In J. J. Alferes and J. Leite, editors, *Proceedings of the 9th European Conference on Artificial Intelligence (JELIA 2004)*, volume 3229 of *LNAI*, pages 200–212. Springer Verlag, sep 2004.

[42] F.Manola and E.Miller. RDF primer. http://www.w3.org/TR/rdf-primer/, February 2004.

[43] H. Garcia-Molina, J. D. Ullman, and J. Widom. *Database System Implementation*. Prentice-Hall, 2000.

[44] A. V. Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *J. ACM*, 38(3):620–650, 1991.

[45] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In R. Kowalski and K. Bowen, editors, *Proceedings of the 5Proceedings Fifth Intl Conference and Symposium*, pages 1070–1080, Cambridge, Mass., 1988. MIT Press.

[46] M. Gelfond and V. Lifschitz. Logic programs with classical negation. In D. Warren and P. Szeredi, editors, *Logic Programming: Proceedings of the Seventh International Conference*, pages 579–597, Cambridge, MA, USA, 1990. MIT Press.

[47] M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Comput.*, 9(3/4):365–386, 1991.

[48] G. Gottlob. Complexity and Expressive Power of Disjunctive Logic Programming. In M. Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium (ILPS '94)*, pages 23–42, Ithaca NY, 1994. MIT Press.

[49] Y. Guo, Z. Pan, and J. Heflin. LUBM: A Benchmark for OWL Knowledge Base Systems. *Journal of Web Semantics*, 3(2–3):158–182, 2005.

[50] C. Gutiérrez, C. A. Hurtado, and A. O. Mendelzon. Foundations of semantic web databases. In A. Deutsch, editor, *Twenty-third ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS 2004), June 14-16, 2004, Paris, France*, pages 95–106. ACM, 2004.

[51] V. Haarslev and R. Mller. Racer: An OWL Reasoning Agent for the Semantic Web. In *Proc. Int'l Wkshp on Applications, Products and Services of Web-based Support Systems (Held at 2003 IEEE/WIC Int'l Conf. on Web Intelligence*, pages 91–95. Society Press, 2003.

[52] T. Hammond. RDF Site Summary 1.0 Modules: PRISM. November 2008.

[53] A. Harth, A. Hogan, R. Delbru, J. Umbrich, S. O'Riain, and S. Decker. Swse: Answers before links! In J. Golbeck and P. Mika, editors, *Semantic Web Challenge*, volume 295 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[54] A. Harth, J. Umbrich, A. Hogan, and S. Decker. YARS2: A Federated Repository for Querying Graph Structured Data from the Web. In *Proceedings of the 6th International Semantic Web Conference 2007 (ISWC2007)*, volume 4825 of *LNCS*, pages 211–224. Springer, 2007.

[55] M. Hepp. Goodrelations: An ontology for describing products and services offers on the web. In *EKAW*, pages 329–346, 2008.

[56] P. Hitzler, R. Sebastian, and M. Krötzsch. *Foundations of Semantic Web Technologies*. Chapman & Hall/CRC, London, 2009.

[57] A. Hogan, A. Harth, and A. Polleres. Scalable authoritative owl reasoning for the web. *International Journal on Semantic Web and Information Systems*, 5(2), 2009.

[58] G. Ianni, T. Krennwallner, A. Martello, and A. Polleres. A Rule System for Querying Persistent RDFS Data. In *The Semantic Web: Research and Applications, 6th European Semantic Web Conference, ESWC 2009, Heraklion, Greece, May 31–June 4, 2009, Proceedings*, volume 5554 of *LNCS*, pages 857–862. Springer, June 2009.

[59] G. Ianni, A. Martello, C. Panetta, and G. Terracina. Efficiently querying RDF(S) ontologies with Answer Set Programming. *Journal of Logic and Computation (Special issue)*, 19(4):671–695, Aug. 2009.

[60] H. M. Jamil. Implementing abstract objects with inheritance in datalogneg. In *VLDB '97: Proceedings of the 23rd International Conference on Very Large Data Bases*, pages 56–65, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[61] M. Kifer. Rules and ontologies in f-logic. In *Reasoning Web*, pages 22–34, 2005.

[62] M. Kifer, G. Lausen, and J. Wu. Logical foundations of object-oriented and frame-based languages. *JACM*, 42(4):741–843, 1995.

[63] A. Kiryakov, D. Ognyanov, and D. Manov. Owlim - a pragmatic semantic repository for owl. In *WISE Workshops*, pages 182–192, 2005.

[64] N. Leone, G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri, and F. Scarcello. The dlv system for knowledge representation and reasoning. *ACM Trans. Comput. Log.*, 7(3):499–562, 2006.

[65] V. Lifschitz, L. R. Tang, and H. Turner. Nested Expressions in Logic Programs. 25(3–4):369–389, 1999.

[66] J. Lu, F. Cao, L. Ma, Y. Yu, and Y. Pan. An Effective SPARQL Support over Relational Databases. In *SWDB-ODBIS*, pages 57–76, 2007.

[67] B. Ludäscher, R. Himmeröder, G. Lausen, W. May, and C. Schlepphorst. Managing semistructured data with florid: A deductive object-oriented perspective. *Inf. Syst.*, 23(8):589–613, 1998.

[68] D. Marin. A Formalization of RDF (Applications de la Logique a la semantique du web). Technical report, Ecole Polytechnique Universidad de Chile, 2004.

[69] D. L. Mcguinness and F. van Harmelen. OWL web ontology language overview. W3C recommendation, W3C, February 2004.

[70] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz. OWL 2 Web Ontology Language: Profiles. World Wide Web Consortium, Working Draft WD-owl2-profiles-20081202, December 2008.

[71] B. Motik, B. C. Grau, I. Horrocks, Z. Wu, A. Fokoue, and C. Lutz (eds.). OWL 2 Web Ontology Language Profiles, June 2009. W3C Candidate Recommendation, available at `http://www.w3.org/TR/2009/CR-owl2-profiles-20090611/`.

[72] B. Motik and R. Rosati. A faithful integration of description logics with logic programming. In *IJCAI*, pages 477–482, 2007.

[73] I. S. Mumick, S. J. Finkelstein, H. Pirahesh, and R. Ramakrishnan. Magic conditions. In *PODS '90: Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pages 314–330, New York, NY, USA, 1990. ACM.

[74] S. Muoz, J. Prez, and C. Gutirrez. Minimal Deductive Systems for RDF. In *ESWC*, pages 53–67, 2007.

[75] I. Niemel. Logic Programming with Stable Model Semantics as Constraint Programming Paradigm. 25(3–4):241–273, 1999.

[76] N. Noy, R. Fergerson, and M. Musen. The knowledge model of protg-2000: Combining interoperability and flexibility. *Lecture Notes in Computer Science*, 1937:69–82, 2000.

[77] P. F. Patel-Schneider, P. Hayes, and I. Horrocks. OWL Web Ontology Language Semantics and Abstract Syntax. Technical report, W3c recommendation, World Wide Web Consortium (W3C), Cambridge, Massachusetts, United States.

[78] T. R. Payne, R. Singh, and K. Sycara. Browsing schedules - an agent-based approach to navigating the semantic web. In *In First Int. Semantic Web Conf*, 2002.

[79] J. Pérez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *International Semantic Web Conference (ISWC 2006)*, pages 30–43, 2006.

[80] P.Hayes. RDF semantics. Technical report, W3C, February 2004. W3C Recommendation.

[81] A. Polleres. From SPARQL to rules (and back). In *Proceedings of the 16th World Wide Web Conference (WWW2007)*, pages 787–796, Banff, Canada, May 2007.

[82] A. Polleres, C. Feier, and A. Harth. Rules with contextually scoped negation. In *ESWC*, pages 332–347, 2006.

[83] A. Polleres, F. Scharffe, and R. Schindlauer. SPARQL++ for mapping between RDF vocabularies. In *6th International Conference on Ontologies, DataBases, and Applications of Semantics (ODBASE 2007)*, volume 4803, pages 878–896, Vilamoura, Algarve, Portugal, Nov. 2007. Springer.

[84] A. Polleres and R. Schindlauer. dlvhex-sparql: A SPARQL-compliant query engine based on dlvhex. In *2nd International Workshop on Applications of Logic Programming to the Web, Semantic Web and Semantic Web Services (ALPSWS2007)*, volume 287 of *CEUR Workshop Proceedings*, pages 3–12, Porto, Portugal, Sept. 2007. CEUR-WS.org.

[85] E. Prud'hommeaux and A. Seaborne (eds.). SPARQL Query Language for RDF, Jan. 2008. W3C Recommendation, available at `http://www.w3.org/TR/rdf-sparql-query/`.

[86] T. C. Przymusinski. Stable Semantics for Disjunctive Programs. 9:401–424, 1991.

[87] A. Rainer. Web service composition under answer set programming. In *KI-Workshop "Planen, Scheduling und Konfigurieren, Entwerfenl" (PuK)*, Winner of the first prize in the [WWW] EEE-Web 2005 Service Composition Contest, 2005.

[88] O. S. released on 2000-12-06. RDF Site Summary (RSS) 1.0. 2006. `http://web.resource.org/rss/1.0/spec`.

[89] F. Ricca, L. Gallucci, R. Schindlauer, T. Dell'armi, G. Grasso, and N. Leone. OntoDLV: An ASP-based System for Enterprise Ontologies. *Journal of Logic and Computation*, Aug. 2008.

[90] F. Ricca and N. Leone. Disjunctive logic programming with types and objects: The dlv$^+$ system. *J. Applied Logic*, 5(3):545–573, 2007.

[91] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, and D. Fensel. Web service modeling ontology. *Applied Ontology*, 1(1):77–106, 2005.

[92] R. Rosati. Dl+log: Tight integration of description logics and disjunctive datalog. In *KR*, pages 68–78, 2006.

[93] P. Simons, I. Niemel, and T. Soininen. Extending and Implementing the Stable Model Semantics. *Artificial Intelligence*, 138:181–234, June 2002.

[94] M. Sintek and S. Decker. TRIPLE - An RDF Query, Inference, and Transformation Language, 2002. International Semantic Web Conference (ISWC).

[95] E. Sirin, B. Parsia, B. Grau, A. Kalyanpur, and Y. Katz. Pellet: A practical OWL-DL reasoner. *Journal of Web Semantics*, 5(2):51–53, 2007.

[96] H. Stuckenschmidt and J. Broekstra. Time - space trade-offs in scaling up rdf schema reasoning. In *WISE Workshops*, pages 172–181, 2005.

[97] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago: a core of semantic knowledge. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 697–706, New York, NY, USA, 2007. ACM Press.

[98] T. Swift. Deduction in ontologies via asp. In *LPNMR*, pages 275–288, 2004.

[99] T.Eiter, G.Ianni, H.Tompits, and R.Schindlauer. Effective integration of declarative rules with external evaluations for semantic web reasoning. In *Proceedings of the 3rd European Semantic Web Conference (ESWC 2006)*, pages 273–287, june 2006.

[100] T.Eiter, G.Ianni, R.Schindlauer, and H.Tompits. Towards efficient evaluation of hex programs. In *Proceedings of 11th International Workshop on Nonmonotonic Reasoning*, pages 40–46, 2006.

[101] T.Eiter, G. Ianni, A.Polleres, and H.Tompits. Reasoning with rules and ontologies. In P.Barahona, F.Bry, E.Franconi, U.Sattler, and N.Henze, editors, *Reasoning Web, Second International Summer School, Tutorial Lectures*, Lissabon, Portugal. Lecture Notes in Computer Science (LNCS), Springer.

[102] H. J. ter Horst. Completeness, decidability and complexity of entailment for rdf schema and a semantic extension involving the owl vocabulary. *Journal of Web Semantics*, 3(2-3):79–115, 2005.

[103] G. Terracina, N. Leone, V. Lio, and C. Panetta. Experimenting with recursive queries in database and logic programming systems. *Theory Pract. Log. Program.*, 8(2):129–165, 2008.

[104] Y. Theoharis, V. Christophides, and G. Karvounarakis. Benchmarking Database Representations of RDF/S Stores. In *Proceedings of the 4th International Semantic Web Conference (ISWC 2005), Galway, Ireland, November 6-10, 2005.*, volume 3729 of *LNCS*, pages 685–701. Springer, 2005.

[105] W3C. The resource description framework., 2006. `http://www.w3.org/RDF/./`.

[106] J. Wielemaker, M. Hildebrand, and J. van Ossenbruggen. Prolog as the Fundament for Applications on the Semantic Web. In *ALPSWS*, 2007.

[107] G. Yang and M. Kifer. Inheritance in Rule-Based Frame Systems: Semantics and Inference. *Journal on Data Semantics*, 7:79–135, 2006.

[108] G. Yang, M. Kifer, and C. Zhao. Flora-2: A rule-based knowledge representation and inference infrastructure for the semantic web. In *CoopIS/DOA/ODBASE*, pages 671–688, 2003.

Acknowledgements

This thesis would not have been possible without the support of many people that I whish to thanks.
First, I would like to express my deepest gratitude to my supervisor, Giovambattista Ianni, who has been a source of enthusiasm and encouragement along the way. I'm very grateful to him for the invaluable assistance and guidance, and for giving me this chance.
I'm pleased to thank Nicola Leone, head of the Department of Mathematics, for giving me the opportunity to work on this interesting topic and for supporting in various ways my growth as a student and researcher.
Big thanks goes to my fellow co-authors Axel Polleres and Thomas Krennwallner for all their help and crucial contribution for this work.
A very special thanks is for the whole Department of Mathematics: to all members of the group I'm very grateful for the cooperative spirit and the excellent working atmosphere. I wish to express my warm thanks to Claudio for taking time out from his schedule to friendly help me.
I cannot express my sincere gratitude to all my friends..who always have been here! Above all, I wish to show my full gratitude to my family, for continuous and unconditional support and especially to my sister for all its understanding.