I4-D14

Exploring and Taming Existence in Rule-based RDF Queries

Project title:	Reasoning on the Web with Rules and Semantics
Project acronym:	REWERSE
Project number:	IST-2004-506779
Project instrument:	EU FP6 Network of Excellence (NoE)
Project thematic priority:	Priority 2: Information Society Technologies (IST)
Document type:	D (deliverable)
Nature of document:	R/P (report and prototype)
Dissemination level:	PU (public)
Document number:	IST506779/Munich/I4-D14/D/PU/a1
Responsible editors:	Tim Furche
Reviewers:	Liviu Badea, Jakob Henriksson
Contributing participants:	Munich
Contributing workpackages:	I4
Contractual date of deliverable:	31 August 2007
Actual submission date:	30 September 2007

Abstract

RDF is an emerging knowledge representation formalism proposed by the W3C. A central feature of RDF are blank nodes, which allow to assert the existence of an entity without naming for it. Despite the importance of blank nodes for RDF, many existing RDF query language have only insufficient support for blank nodes. We propose a query language for RDF, called RDFLog, with extensive blank node support. The evaluation of RDFLog may be reduced to the evaluation of Datalog. This allows to apply standard database technology to querying RDF. Our Experimental evaluation shows that our implementation scales well, even for large data sets. The core feature of the reduction is Skolemisation and an new form of un-Skolemisation. In contrast to previous definitions of un-Skolemisation our un-Skolemisation has desirable symmetric properties to those of the Skolemisation. We define a hierarchy of syntactical restrictions of RDFLog with lower expressivity but better complexity, thereby showing the computational costs of blank nodes. We provide both an operational and a denotational semantics of RDFLog and show its soundness and completeness. Finally we discuss a notion of redundancy free answers, called lean answers, and state their complexity and an operational semantics.

Keyword List

RDFLog, blank nodes, existence, RDF, querying, rules

Project co-funded by the European Commission and the Swiss Federal Office for Education and Science within the Sixth Framework Programme.

Exploring and Taming Existence in Rule-based RDF Queries

Clemens Ley[1], **François Bry**[1], **Tim Furche**[1], **Benedikt Linse**[1]

[1] Institute for Informatics, University of Munich, Germany
http://pms.ifi.lmu.de

30 September 2007

Abstract
RDF is an emerging knowledge representation formalism proposed by the W3C. A central feature of RDF are blank nodes, which allow to assert the existence of an entity without naming for it. Despite the importance of blank nodes for RDF, many existing RDF query language have only insufficient support for blank nodes. We propose a query language for RDF, called RDFLog, with extensive blank node support. The evaluation of RDFLog may be reduced to the evaluation of Datalog. This allows to apply standard database technology to querying RDF. Our Experimental evaluation shows that our implementation scales well, even for large data sets. The core feature of the reduction is Skolemisation and an new form of un-Skolemisation. In contrast to previous definitions of un-Skolemisation our un-Skolemisation has desirable symmetric properties to those of the Skolemisation. We define a hierarchy of syntactical restrictions of RDFLog with lower expressivity but better complexity, thereby showing the computational costs of blank nodes. We provide both an operational and a denotational semantics of RDFLog and show its soundness and completeness. Finally we discuss a notion of redundancy free answers, called lean answers, and state their complexity and an operational semantics.

Keyword List
RDFLog, blank nodes, existence, RDF, querying, rules

Contents

Overview of this Deliverable

One important difference of conventional data and Semantic Web data lies in the ability of RDF and similar knowledge representation formalisms suggested for the Semantic Web to express purely existential (but unnamed) data.

Though this is a discriminating and often used feature of Semantic Web data, *query answering* in presence of existential information has received little attention. This issue becomes particularly interesting when considered in combination with rules, where existential information may not only occur in the base (extensional, factual) data, but may also be inferred (in technical terms: occur in rule heads).

In this deliverable, we propose the first complete treatment of RDF's existential information approach, called blank nodes, over the whole range of (logic-based) query languages traditionally used in databases.

In addition to the theoretical results presented in this deliverable, we have also created an operational semantics of blank nodes in rule heads (cf. Chapter 6). This operational semantics has been implemented in a first prototype, which is available from `http://rdflog.com`. There we will continuously update the prototype as well as provide experimental results (preliminary results indicate, that the approach is, though much more expressive, very competitive with existing SPARQL engines).

In the future, we plan on integrating the discussed technology in the versatile Xcerpt query language (cf. `http://xcerpt.org`).

Chapter 1

Introduction

1.1 Motivation

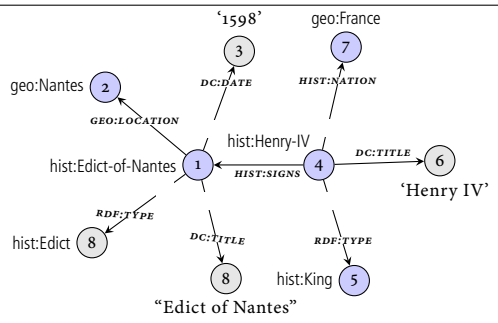
Naming people, concepts, or things is a problem as old as language. We need names to refer to entities when we exchange data. This is true also for the Web, where RDF has become the W3C data interchange standard. However, "good" names are notoriously difficult: they must be unique and shared (or translated). Therefore we often talk about things by its properties and relations rather than by its name. RDF offers anonymous or "blank" nodes for just this purpose. This facility is, arguably, one of the main novelties of RDF. Its use is rather common, in particular for representing complex structures such as lists, sets, sequences, n-ary relations, or reified statements.

For the most part, we can safely think of RDF as relational data or, in terms of logic, as ground facts. Blank nodes, however, introduce merely existential knowledge: We might know that "there is an edict signed in Nantes by some French king" without knowing the actual identity of the edict or the king. At first, such an addition may look fairly harmless. However, the presence of blank nodes raises a number of questions that do not occur when considering only named nodes (or, indeed, relational data): What if there are two blank nodes with the same properties (as statements 1 and 2 in Figure 1)? What if there is a named node with the same (or more) properties than a blank node? What if there are two blank nodes with overlapping but not identical properties?

All of these questions are in essence about what constitutes identity in presence of blank nodes or, more practically, when is a given blank node a duplicate of another node and thus redundant. Blank nodes are RDF's answer to the long discussed question of value-based or extensional identity (as in the relational model) versus surrogate or object identity: Named nodes in RDF have a surrogate identity by way of their associated URI: there can be two nodes with the same properties but different identity. Blank nodes, in contrast, have an extensional identity: if they have the same properties they are considered identical ([Note: this is somewhat imprecise, since it does not mention properties referring to other blank nodes]) . RDF puts its own twist on the issue as it considers also a blank node with a subset of properties of another node as redundant.

At first glance, this treatment of blank nodes in RDF may seem arbitrary. However, it turns out that, if we consider blank nodes as existential variables and RDF triples as logical atoms, we arrive at a view

Figure 1 RDF graph on the 'Edict of Nantes'



of **RDF data as formulas** (possibly infinite formulas with infinite variables but restricted to conjunction and existential quantification). Our first contribution, is a semantics for simple RDF that is fully aligned with the RDF model theory but based on this observation. The main advantage of this semantics is that it clarifies the meaning of blank nodes in terms familiar from standard logics. Also, our semantics is entirely transformational, in contrast to and previous logic- or graph-based approaches we do not need any special forms of interpretation, model, or entailment.

Given a clear semantics, we can start tackling the effect of blank nodes on querying: Say we want to find out where French edicts revoking rights of religious tolerance have been signed. From the data, we know that "some edict was signed in Nantes". We also know that "some French edict is about religious tolerance". It is possible that in both cases we are speaking about the same edict, but we cannot be certain of that—indeed, the latter statement is about the Edict of Fontainbleau. In RDF querying and the remainder of this article, we only consider *certain answers* (in contrast to possible answers). The second effect occurs when considering also views or rules. We might, e.g., know that "each edict signed in France during absolutism must be signed by some French king" without knowing exactly which king (e.g., since we are lacking exact dates for the edict or for the reign of the various kings). When we query for all edicts signed by some French king, this information suffices to return also all French edicts signed during the times of absolutism. In this way, *inferred existential information* (or blank nodes in rule heads) allows us to match existential queries. We might also be interested in the actual kings, which brings us to the final issue: *blank nodes in answers* (or intermediary answers). Let us, for the moment assume, that an answer is again an RDF graph, for instance, the graph containing signatories of French edicts connected to the edicts they signed. For the data of Figure 1, we obtain two signatories for the "edict signed in Nantes". However, all the information about the unnamed (blank node) signatory is already implied by the information about Henry IV. Returning such redundant information in answers serves no purpose. Worse, a naïve computation of redundant answers in presence of blank nodes in rule heads can easily lead to infinite answers where a non-redundant answer is finite.

To answer the first two issues,

1. we extend **Datalog with blank nodes** (in other words: existential quantifiers in facts and rule heads) and call that extension, restricted to RDF data, RDFLog;

2. we define a **closed and generic denotational semantics** for RDFLog (i.e., a semantics where input and output of a query program are RDF graphs and which is oblivious to consistent renaming

of resource identifiers, the constants of RDF) based on RDF entailment. The semantics is fully aligned with the RDF semantics (in particular, for two equivalent input graphs, equivalent results are returned);

3. we define a (two-way) transformation from RDFLog programs to logic programs without blank nodes and show that this transformation yields a sound and complete operational semantics for RDFLog. The first transformation is a Skolemisation. The second transformation is a novel form of **un-Skolemisation**, which has desirable properties symmetric properties to those of Skolemisation;

4. we define a **hierarchy of language restrictions** for RDFLog and relate these restrictions, again using the above transformation, to restrictions for universal logic programs. This yields a hierarchy of increasingly more expressive languages together with their **complexity**;

5. beyond the theoretical results, the approach demonstrates how we can realize these different levels using **existing technology** for universal logic programs (e.g., SQL, Datalog or Prolog engines). **Experimental evaluation** validates that the presented approach is clearly competitive with existing special purpose RDF engines. The transformation overhead is, as expected, very small and, in most cases, negligible;

6. finally we define a notion of redundancy free answers called **lean answers**. We show that these kinds of answers are computationally more expensive than non-lean answers and give an operational optimisation.

Part I

Preliminaries

Chapter 2

Logic

In this deliverable we use logics for three purposes:

(a) as an abstraction of RDF

(b) as the syntax of the RDF query language RDFLog we propose,

(c) to formulate a semantics for RDFLog.

We show in the next chapter that logic is well suited as an abstraction for RDF. It is possible to translate an RDF graph to logic in such a way that both validity and entailment are preserved. The second point is motivated as we want to use techniques from logic programming to query RDF. It is common to use variants of logical syntaxes for logic programming language. The third point might be more controversial. It is desirable for query languages to produce output in the same formalism as the input (this is referred to as *closure*). This is a common approach for database languages which are not Turing complete.

With Turing complete languages the problem arises that the semantics of programs might be infinite. To avoid infinite formulas the semantics of logic programming is described in terms of models. Therfore one could state that logic programming is not closed. Nonetheless, as there is a direct correspondence between Herbrand models and formulas: A Herbrand Model is usually characterised by a (possibly infinite) set of atomic formulas.

In the case of query languages for RDF the situation is more complicated. The abstraction of RDF presented in Chapter 3 modes blank nodes as existential variables. As the same blank node may occur in many (possibly even infinitely many) atomic RDF statements, the scope of an existential quantifier might range over an infinite number of atomic formulas. Therfore we consider formulas of infinite length in this work.

It should be pointed out that there is also the possibility to model the semantics of RDFLog in terms of models. We provide a translation from formulas to interpretations in Chapter 3. Any reader with ontological considerations is therefore friendly advised to think of infinite models instead of infinite formulas.

2.1 Terms and Formulas

Like in formal language theory [45, 39] a logical language are defined over a set called alphabet.

Definition 1 (Logical Symbols, Alphabet). The *logical symbols* is the set consisting of the following symbols:

- a countably infinite set *Var* of *variables*,
- the symbols $\neg$ for negation , $\bigwedge$ for conjunction, $\exists$ for existential quantification, (and) as parentheses, and
- a set A called *alphabet* consisting of the disjoint sets $Pred_A$ and $Funk_A$ called *predicate symbols* and *constant symbols*. Each symbol s in an alphabet is associated with a natural number $n \geq 0$ called the arity of s. If a function symbol has arity 0 then it is called a *constant symbol*.

There is a conceptual distinction in logic between entities with meaning and entities with truth [67, 66]. The entities which have meaning are called 'terms' and the entities with truth are called formulas.

Definition 2 (Term). If f is an n-ary function symbol and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a *term*. If f is 0-ary then we omit the brackets and say that f is a *constant*. We call f *the* function symbol of the term $f(t_1, \ldots, t_n)$.

To define the syntax of formulas we need some notation. We denote the sequence $[x_1, x_2, \ldots]$ by $\bar{x}$. We sometimes treat sequences as sets when it is convenient. For example we write $x \in \bar{x}$ to denote that x is an element in the sequence $\bar{x}$. We do not always specify the length of a sequence. In general a sequence may also be empty. If f is a function and $\bar{x} = [x_1, x_2, \ldots]$ is a sequence then $f(\bar{x})$ denotes the sequence $[f(x_1), f(x_2), \ldots]$.

Definition 3 (Formula). A *formula* is defined by a finite application of the following rules:

- If p is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms then $p(t_1, \ldots, t_n)$ is a *(atomic-) formula*.
- If φ is a formula then $\neg(\varphi)$ is a formula.
- If Φ is a possibly infinite set of formulas then $\bigwedge_{\varphi \in \Phi} \varphi$ is a formula.
- If φ is a formula and $\bar{x}$ is a possibly infinite sequence of variables then $\exists \bar{x}(\varphi)$ is a formula.

We use the following abbreviations: We write $\bigvee_{\psi \in \Phi} \psi$ for $\neg \bigwedge_{\psi \in \Phi} \neg\psi$. If $\Phi = \{\psi_1, \ldots, \psi_n\}$ is finite we write $\psi_1 \wedge \ldots \wedge \psi_n$ instead of $\bigwedge_{\psi \in \Phi} \psi$ and $\psi_1 \vee \ldots \vee \psi_n$ instead of $\bigvee_{\psi \in \Phi} \psi$. We sometimes write $\bigwedge \Phi$ ($\bigvee \Phi$) instead of $\bigwedge_{\varphi \in \Phi} \varphi$ ($\bigvee_{\varphi \in \Phi} \varphi$). A universal quantifier $\forall$ may be used to write $\forall \bar{x}(\varphi)$ for $\neg \exists \bar{x}(\neg\varphi)$.

Finally, we use the following precedences to avoid brackets: $\neg$ binds stronger than $\bigwedge, \bigvee, \wedge$ and $\vee$, which in turn bind stronger than the quantifiers $\forall$ and $\exists$. A scope of a quantifier is as small as possible, unless the quantifier is followed by a dot, in which case the scope is as large as possible.

2.2 Interpretations and Models

The notion of interpretation is used assign a meaning to terms and a truth value to formulas. If a formula φ is true in an interpretation$I$ then I is called a *model* of φ.

Definition 4 (Interpretation). An *interpretation* I is a triple (D, i, β) where D is a set called *domain* of I and i is a mapping from every n-ary function symbol f (predicate symbol p) to an n-ary function $f^i : D^n \to D$ (an n-ary relation $p^i \subseteq D^n$) and $\beta : Var \to D$ is a variable assignment.

Throughout this deliverable we sometimes identify functions with their graphs. This allows us to extend functions in a simple way. We also sometimes treat sequences as sets when the order is irrelevant.

Let $I = (D, i, \beta)$ is an interpretation, x a variable and $d \in D$. Then we denote by $I\frac{d}{x}$ the interpretation defined by $(D, I, \beta \cup \{(x, d)\})$.

Definition 5 (Validity). Let $I = (D, i, \beta)$ be an interpretation. The interpretation $I(t)$ of a term t is defined as:

- If $t \equiv f(t_1, \ldots, t_n)$ then $I(t) = f^i(I(t_1), \ldots, I(t_n))$.

A formula φ is called valid in an interpretation I (denoted $I \models \varphi$) if the following is true:

- If $\varphi \equiv p(t_1, \ldots, t_n)$ then $(I(t_1), \ldots, I(t_n)) \in p^i$.
- If $\varphi \equiv \neg\psi$ then it does not holds that $I \models \psi$.
- If $\varphi \equiv \bigwedge_{\psi \in \Phi} \psi$ then it holds for all $\psi \in \Phi$ that $I \models \psi$.
- if $\varphi \equiv \exists \bar{x} \psi$ then there exists a total variable assignment $\beta : \bar{x} \to D$ such that $I\frac{\beta(\bar{x})}{\bar{x}} \models \psi$.

The basic relation between formulas is entailment.

Definition 6 (Entailment). We say that a formula φ entails a formula ψ (denoted $\varphi \models \psi$) if every model of φ is also a model of ψ. This definition is extended to sets of formulas in the obvious way.

2.3 Normal Forms and Basic Results

This section contains some notions used in the following chapters.

Definition 7 (Prenex Normal Form). A formula is in prenex normal form if all quantifiers are left of the leftmost predicate.

We call a formula *universal* if it is in prenex normal form and contains only universal quantifiers.

We often make use of the substitution Lemma. We ommit the easy proof by induction as it can be found in many basic books on logic such as [28]

Lemma 1 (Substitution Lemma). Let Φ be a set of sentences and φ a formula and $\{\bar{x}/\bar{t}\}$ a substitution. Then it holds that

$$I\frac{I(\bar{t})}{\bar{x}} \models \varphi \quad \Leftrightarrow \quad I \models \varphi\{\bar{x}/\bar{t}\}$$

Chapter 3

RDF

The *Resource Description Framework* [54, 57] has been developed by the W3C as a simple data and metadata representation language. Together with XML and URIs it is expected to form the basis of the Semantic Web. The basic modelling concept of RDF is strikingly simple: statements are formed of *triples* consisting of a subject, a predicate and an object. Subjects, predicates and objects are drawn from the infinite sets of URIs (U), literals (L), and blank nodes (B). URIs are used in RDF to uniquely identify an object or concept. In contrast, literals are merely values without object identity. Using a blank node amounts to a statement that something exists, and that it full fills the relations given by the triples in which it occurs. A collection of such triples forms an RDF graph. An RDF graph can be seen both as a set of triples or as a logical formula.

In this chapter we first recall the definition of the RDF syntax and semantics given by the W3C in the standardising documents [53, 40]. These documents define RDF to be a labelled graph. In addition to these graphs a notion of interpretations and validity is defined. We discuss the advantages and shortcomings of these definitions. As we want to use logic programming to query RDF we show how an RDF graph can be translated to a logical formula and a RDF interpretation to a logic interpretation such that validity is preserved. We also dicuss how RDF graphs themselves may be modelled as interpretations. Such a view is commonplace in database theory. We show how this view allows to reduce RDF entailment to validity using Hayes' interpolation lemma [40]. We finally discuss a notion called leanness which is a form of redundancy freeness.

3.1 RDF Syntax and Semantics

In this section we recall the syntax of RDF graphs as defined in [53]. In Section 3.2 and Section 3.3 we will give an equivalent definition of an RDF graph as a formula or an interpretation.

The nodes of an RDF graph are elements of the following sets.

Definition 8 (URI Reference, Literal, Bank Node [53])**.** A *URI reference* is a string that would produce a valid URI. We denote the set of URI references by U.

An *RDF literal* is a string. A literal may have a *language tag* (again a string) or a *datatype URI* (a URI reference) associated with it. We denote the set of literal by L.

A *blank node* is an element of some infinite set B. We assume that U, L and B are disjoint. The set $U \cup L$ is called a *vocabulary*.

An atomic statement in RDF is called a triple. A set of such triples is an RDF graph.

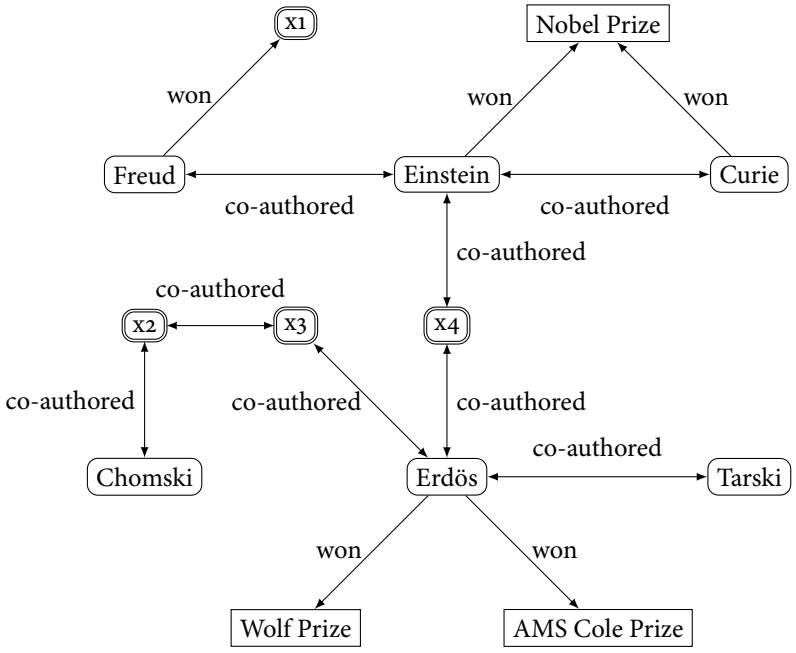
Definition 9 (RDF Triple, RDF Graph [53])**.** An *RDF triple* is a triple

$$(s, p, o) \in (U \cup B) \times U \times (U \cup L \cup B)$$

An *RDF graph* is a set of RDF triples.

Figure 2 shows an RDF graph which describes co-author graph. Throughout this document literals are depicted as rectangles, URIs and blank nodes have rounded corners, where URI references have a single border and blank nodes have a double border. By misuse of notation we often name URIs references and blank nodes by strings instead of URIs if this does not lead to confusion.

Figure 2 A co-author graph in RDF



RDF graphs may be used to make statements about the world. This statement is considered true, if the world can be seen as an interpretation of the world. Hayes put it this way: 'The basic intuition of model-theoretic semantics is that asserting a sentence makes a claim about the world: it is another way of saying that the world is, in fact, so arranged as to be an interpretation which makes the sentence true. In other words, an assertion amounts to stating a constraint on the possible ways the world might be.' [40] These interpretations are defined as follows.

Definition 10 (Simple RDF Interpretation [40])**.** Let V be a vocabulary. A *simple RDF interpretation* I is defined by a tuple $(\mathsf{IR}, \mathsf{IP}, \mathsf{IEXT}, \mathsf{IS}, \mathsf{IL}, \mathsf{LV})$ consisting of

- a non-empty set IR of resources, called the domain or universe of I,
- a set IP, called the set of properties of I,
- a mapping IEXT from IP into the power-set of $\mathsf{IR} \times \mathsf{IR}$,
- a mapping IS from URI references in V into $\mathsf{IR} \cup \mathsf{IP}$,
- a mapping IL from typed literals in V into IR, and
- a distinguished subset LV of IR, called the set of literal values, which contains all the plain literals in V.

The following definition resembles the definition of validity.

Definition 11 (Semantic Conditions for Ground Graphs [40]). Let E be a literal, a URI reference, an *RDF*triple of an RDF graph. Then we define

- if E is a plain literal 'a' in V then $\mathsf{I}(E) = a$.
- if E is a plain literal 'a'@t in V then $\mathsf{I}(E) = (a, t)$.
- if E is a typed literal in V then $\mathsf{I}(E) = \mathsf{IL}(E)$.
- if E is a URI reference in V then $\mathsf{I}(E) = \mathsf{IS}(E)$.
- if E is a ground triple (s, p, o) then $\mathsf{I}(E) = \mathit{true}$ if
  - s, p and o are in V,
  - $\mathsf{I}(p)$ is in IP, and
  - $(\mathsf{I}(s), \mathsf{I}(o))$ is in $\mathsf{IEXT}(I(p))$,

  otherwise $\mathsf{I}(E) = \mathit{false}$.
- if E is a ground RDF graph then $\mathsf{I}(E) = \mathit{false}$ if $\mathsf{I}(E') = \mathit{false}$ for some triple E' in E, otherwise $\mathsf{I}(E) = \mathit{true}$.

In the following validity for blank nodes is defined. Observe that an RDF graph containing a blank node b is valid if there exists a mapping $\mathsf{A} : B \to \mathsf{IR}$ to interpret blank nodes. This gives rise to the similarity between blank nodes and existential variables.

Definition 12 (Semantic conditions for blank nodes [40]). Let A be a mapping from B to the universe IR of I, and define $[\mathsf{I}+\mathsf{A}]$ to be an extended interpretation which is like I except that it uses A to give the interpretation of blank nodes. Define $blank(E)$ to be the set of blank nodes in E. Then the above rules can be extended by the following:

- If E is a blank node and $A(E)$ is defined then $[\mathsf{I}+\mathsf{A}](E) = \mathsf{A}(E)$
- If E is an *RDF*graph then $I(E) = \mathit{true}$ if $[\mathsf{I}+\mathsf{A}'](E) = \mathit{true}$ for some mapping A' from $blank(E)$ to IR, otherwise $\mathsf{I}(E) = \mathit{false}$.

Entailment between RDF graphs is defined as for logical formulas.

Definition 13 (RDF Entailment [40]). *I* satisfies *E* if $\mathsf{I}(E) = true$, and a set S of RDF graphs *(simply) entails* a graph *E* if every interpretation which satisfies every member of *S* also satisfies *E*.

As there are an infinite number of interpretations of an RDF graph, this definition does not directly induce an computable algorithm. Such an algorithm is implied by the interpolation lemma [40] which uses the notions of instance and subgraph:

Definition 14 (Subgraph, Instance [40]). A *subgraph* of an RDF graph $g = \exists \tilde{x}. \bigwedge \Phi$ is an RDF graph $\exists \tilde{y}. \bigwedge \Psi$ where $\Psi \subseteq \Phi$.

Let σ be a substitution. An *instance* of an RDF graph g is an RDF graph $g\sigma$. An instance of an RDF graph g is a real instance if $subs$ maps a blank node in g to a constant.

Lemma 2 (Interpolation Lemma [40]). An RDF graph g entails an RDF graph h ($g \models h$) iff a subgraph of g is an instance of $rdfgraph$.

A direct consequence of the interpolation Lemma in [40] is the subgraph Lemma

Lemma 3. An RDF graph entails any of its subgraphs.

It has been shown that RDF entailment is NP complete [35] and that it is polynomial when the entailed RDF graph is a tree [8]. The same authors also show the equivalence between RDF entailment, graph homomorphisms and constraint networks. This makes results of the constraint satisfaction community, such as hypertree decomposition [34], accessible for RDF entailment.

An interesting semantic notion in the context of RDF is the notion of a lean RDF graph. Leanness captures in some sense that an RDF graph is free of redundancy. The precise definition is

Definition 15 (Lean). An RDF graph g is *lean*, if no instance of g is a subgraph of g.

This notion, indeed, captures the intuition that there may be no proper subgraph with the same entailment as the following proposition states (which follows directly from interpolation and subgraph lemma above).

Proposition 1. Let g be an *RDF*-graph and σ a substitution consistent with g. Then

$$g \supseteq_{RDF} g\sigma \quad \Longrightarrow \quad g\sigma \models\!\!\!\!| \; g$$

Two RDF graphs are called *isomorphic* if they are equal up to renaming of blank nodes. Observe that isomorphism implies bi-entailment, but not vice versa.

Observe that the RDF semantics defines two different notions of the composition of RDF graph. The *union* of two RDF graphs is the union in a set theoretic sense. The *merge* of two RDF g and h graphs is the union of the RDF graphs g' and h' where g' is isomorphic to g and h' is isomorphic to h and g' and h' have no blank nodes in common.

3.2 RDF Graphs as Formulas

In this thesis we propose a query language for RDF based on logic programming. We want to consider an RDF graph as the facts of such a program. We show in this chapter that this approach integrates well

with the definition of the semantics of RDF: an RDF graph can be translated to a formula and an RDF interpretation can be translated into a logic interpretation such that validity is preserved.

The translation of an RDF graph to a formula is straightforward. A URI or a literal is translated to a constant. A blank node is translated to an existential variable. A triple is translated to an atomic formula. An RDF graph g is translated to the conjunction of all atomic formulas representing triples in g.

When it comes to interpreting theses formulas using a logic interpretation a problem arises: In logic one usually considers the sets of predicate symbols to be disjoint from the set of function symbols. This is not the case for RDF. To be able to use the standard definition of interpretation we prefix all predicate symbols in an RDF graph with '*pred:*' before the translation.

Definition 16 (RDF Alphabet, Translation of an RDF Graph)**.** The *RDF predicate symbols* $Pred_{RDF}$ is the set $\{pred{:}u \mid u \in U\}$. The *RDF function symbols* $Funk_{RDF}$ is the set $U \cup L$. The *RDF alphabet* A_{RDF} is the set $Pred_{RDF} \cup Funk_{RDF}$

Let g be an RDF graph. An existentially quantified sentence φ is *the translation of* g iff

- there is a bijection f from the blank nodes in g to the variables occurring in φ
- g contains the triple (s, p, o) iff φ contains the conjunct $pred : p(s, o)$.

It is easy to see that the translation of an RDF graph to a formula is unique up to renaming of variables. We therefore speak of *the* translation of an RDF graph. Observe that as an RDF graph may have an infinite number of triples. a translation of an RDF graph may have an infinite number of conjuncts.

We now show how an RDF interpretation can be translated to a logic interpretation.

Definition 17 (Translation of an RDF Interpretation)**.** Let $I = (\mathsf{IR}, \mathsf{IP}, \mathsf{IEXT}, \mathsf{IS}, \mathsf{IL}, \mathsf{LV}, \mathsf{A})$ be an RDF interpretation. A logic interpretation $J = (D, j, \beta)$ is the *translation of* I if

- $D = \mathsf{IR}$,
- $p^j = \mathsf{IEXT}(\mathsf{IS}(p'))$ where $p = pred : p'$ and $p \in Pred_{RDF}$, and
- $t^j = I(t)$ if $t \in Funk_{RDF}$.
- $\beta = \mathsf{A}$

The following proposition stated that our translation preserves validity.

Proposition 2. Let g be an RDF graph and let g^{tr} be its translation to a formula. Let I be a RDF interpretation and let I^{tr} be its translation to a logic interpretation. Then it holds that I is a model of g iff I^{tr} is a model of g^{tr}.

The easy proof by induction is omitted.

3.3 RDF Graphs as Interpretations

In this section we show that an RDF graph can also be seen as an interpretation. We will show that this view admits to reformulate RDF entailment in model theoretic and logical terms. In addition we will see that the semantics of a logic program is characterised by its minimal Herbrand model. The semantics of the language RDFLog is formulated in terms of RDF graphs. The following translation allows us to compare both semantics.

Definition 18. Let $g = \exists \bar{x}. \bigwedge \Phi$ be an rdf graph over alphabet A. Let T be the set of terms occurring in g. An interpretation $I = (D, I, \beta)$ over A is called *related* to g if

- there is a bijection $f : D \to T$ such that $I(f(d)) = d$ for all $d \in D$.
- $f^i = f$ for all $f \in Funk_A$.
- $p^i = \{(d_1, d_2) \in D \times D \mid p(f(d_1), f(d_2))$ is a conjunct in $g\}$

Let I, J be two interpretations related to an RDF graph g. Observe that in this case I and J are isomorphic and therefore elementary equivalent (they are models of exactly the same sentences, compare [44]). This allows us to speak of *the* interpretation related to an RDF graph g which we denote by I_g.

We now show how this view allows as to reformulate entailment in terms of validity.

Lemma 4. Let g, h be an RDF graphs. Then it holds that if the interpretation I_g related to g is a model of h then g entails h.

Proof. Let $h = \exists \bar{x} \,.\, \bigwedge \Phi$ where Φ is a set of binary atomic formulas. Let D be the domain of I_g. As $I_g \models h$ it holds that there is a mapping $\beta : \bar{x} \to D$ such that $I_g \frac{\beta(\bar{x})}{\bar{x}} \models \bigwedge \Phi$.

Let f be the bijection from D and to the set of terms T occurring in g. Observe that $f(\beta(\bar{x}))$ is a sequence of terms in g. As $I_g(f(d)) = d$ it holds that $I_g \frac{I_g(f(\beta(\bar{x})))}{\bar{x}} \models \bigwedge \Phi$ and by the substitution Lemma that 1 $I_g \models \bigwedge \Phi\{\bar{x}/f(\beta(\bar{x}))\}$.

We define the substitution σ as $\{\bar{x}/f(\beta(\bar{x}))\}$. Let $p(a, b) \in \Phi\sigma$ where a and b are terms. By Hayes interpolation Lemma (Lemma 2) we need to show that $p(a, b)$ is a conjunct of h. This is the case as $I_h \models p(a, b)$ iff $(a^i, b^i) \in p^i$ and this implies that $p(a, b)$ is a conjunct of g by definition. □

In addition state the following lemma which shows how RDF entailment relates to the substructure relationship. The proof is omited.

Proposition 3. Let g, h be RDF graphs and let I_g, I_h be the interpretations related to g and h. Then it holds that $g \models h$ iff there is a homomorphism from I_h to I_g.

Chapter 4

Logic Programming

This section is not meant to be an introduction to logic programming. Comprehensive introductions to logic programming and Prolog can be found in [73, 62, 56, 25]

Instead we compare logic programming to theorem proving and state why it is more efficient in terminating cases.

4.1 Logic Programming vs. Theorem Proving

4.1.1 The Logic Programming Problem

Logic programming can be seen as an instance of theorem proving. The aim of theorem proving is to solve the following problem efficiently.

- Given a set of formulas Φ and a formula φ. Decide whether $\Phi \models \varphi$

There are several systems to solve this problem [60, 6, 68]. Nonetheless theorem proving is highly inefficient in many cases and therefore not suitable as a programming language. This justifies the search for a syntactical restriction of first order logic which is still Turing complete but which has a more efficient calculus in terminating cases. The syntactical restriction used in logic programming is that both Φ and φ are Horn clauses.

Definition 19 (Literal, Clause, Horn Clause)**.** A *literal* is an atom or the negation of an atom. A *clause* is a disjunction of literals. A *Horn clause* is a clause with at most one positive literal. A Horn clause is called *positive* if it contains a positive literal and *negative* otherwise.

We usually denote the positive Horn clause $\neg a_1 \vee \ldots \vee \neg a_n \vee a$ by $a_1 \wedge \ldots \wedge a_n \to a$. This is justified because both formulas are equivalent.

The observation of logic programming is that if Φ consists of Horn clauses, and φ is a closed formula of the form $\exists \tilde{x}.\varphi_1 \wedge \ldots \wedge \varphi_n$ where the φ_i are atoms, then $\Phi \models \varphi$ may be decided efficiently if the computation terminates. We refer to this instance of the theorem proving problem to the *logic programming problem*. In the following sections we discuss why this problem may be solved more efficiently than theorem proving in many cases.

Therefore we define

Definition 20 (Logic Program, Query)**.** A *logic program* is a finite set of universal positive Horn clauses. A *query* is a universal negative Horn clause.

First observe that deciding entailment may be reduced to unsatisfiability: it holds that $\Phi \models \varphi$ iff $\Phi \cup \{\neg\varphi\}$ is unsatisfiable. Therefore if we can decide unsatisfiability of a set of Horn clauses efficiently, we have obtained an efficient procedure to decide the logic programming problem.

To decide un-satisfiability requires in the general case to check for every interpretation I that I is not a model of $\Phi \cup \{\neg\varphi\}$. As there is an infinite number of interpretations for a given alphabet, this is surely un-feasible.

4.1.2 Term Models

A construction of Henkin can be used to solve this problem. Henkin wanted to find an simpler proof for Gödels proof of the completeness of first order logic with respect to his functional calculus [33]. Henkin shows that

$$\text{if } \Phi \text{ is consistent, then it has a model.} \qquad (\star)$$

It can be easily seen that this statement implies the completeness of first order logic (see [28] Chapter V).

To show $(\star)$, Henkin first shows that every consistent set of first order sentences Φ, which satisfies certain properties with respect to negation (i) and existential quantification (ii) has a model [41]. In addition it is possible to extend Φ in such a way that the extension satisfies (i) and (ii). It should be noted though that this extension adds a countably infinite number of formulas to Φ.

We first show informally how to prove $(\star)$ for a set Φ which satisfies (i) and (ii) and then argue how it is possible to avoid the infinite extension. We show $(\star)$ by constructing for a given set of formulas Φ a model $I^\Phi = (D, i, \beta)$. The idea of the construction of I^Φ is to interpret every term by itself and to interpret an predicate p as true iff $\Phi \vdash p$ (where $\vdash$ is some sound calculus for first order logic). Models of this kind are referred to as *term models*.

It might be tempting to choose the domain D to be the set of ground terms. But, there is a problem if Φ contains equality. Consider the case that $t_1 = t_2 \in \Phi$ for distinct terms t_1, t_2. As t_1, t_2 are distinct it holds that $I^\Phi(t_1) = t_1 \neq t_2 = I^\Phi(t_2)$ and thus $I^\Phi \not\models t_1 = t_2$. Therefore I^Φ is not a model of Φ. This can be avoided by defining D to be the set of equivalence classes induced by the equivalence relation defined as $t_1 \sim t_2$ iff $\Phi \vdash t_1 = t_2$.

4.1.3 Herbrand Models

Therefore, the first restriction we use is to assume that Φ does not contain equality. In this case every equivalence class in the domain of I^Φ consists of a single element. We identify the singleton equivalence class $[t]_\sim$ with t and therefore term models simplify to Herbrand models

Definition 21 (Herbrand Universe, Base, and Interpretation)**.** Let A be an alphabet. The *Herbrand universe* U_A of A is the set of all ground terms build from symbols in A. The Herbrand base B_A of A is the set of all ground atoms which can be formed from the symbols in A.

An interpretation $I = (D, i, \beta)$ over an alphabet A is an Herbrand interpretation if $D \supseteq U_A$ and $a^i = a$ for all $a \in A$.

If P is a logic program over alphabet A, then we denote the Herbrand universe over A by U_P and Herbrand base over A by B_P.

Observe that the definition of an Herbrand model does not say anything about the interpretations of predicate symbols. Thus a Herbrand interpretation is fixed for some alphabet A if we fix the interpretation for the predicate symbols. Therefore we often identify an Herbrand interpretation I with the set of atoms which are true in I.

The restriction to equality free formulas simplifies the models we are interested in but it has no advantages with respect to computability. We still need to extend the set of formulas Φ to an infinite set to compute a model. The next Proposition shows that this is not necessary if we restrict Φ to universal Horn clauses. We formulate it in a more general form than needed for logic programming as we shall need this general form later on.

Proposition 4. Let Φ be a consistent set of universal positive Horn clauses, Ψ a set of atomic formulas, and $\bar{x}$ all variables occurring in Ψ. Let $I^\Phi = (D, i, \beta)$ be a Herbrand interpretation over A such that $I^\Phi \models a$ iff $\Phi \models a$ for all atoms a. Then the following is equivalent:

(a) $\Phi \models \exists \bar{x}. \bigwedge \Psi$

(b) $I^\Phi \models \exists \bar{x}. \bigwedge \Psi$

(c) there is a sequence $\bar{t} \in U_P$ such that $\Phi \models \bigwedge \Psi\{\bar{x}/\bar{t}\}$.

Proof. The proof is an extention of the proof for Proposition XI.2.7 in [28] It is trivial that (c) implies (a). To see that (a) implies (b) observe that $I^\Phi \models \Phi$ (which is proved in [28] XI.2.4).

Finally to show that (b) implies (c) assume that (b) is true. As I^Φ is a term model, there is a sequence $\bar{t}$ of terms such that $I^{\Phi\,\frac{\bar{t}}{\bar{x}}} \models \bigwedge \Psi$. As $I^\Phi(t) = t$ for all terms t it follows from the substitution Lemma that $I^\Phi \models \bigwedge \Psi\{\bar{x}/\bar{t}\}$. Therefore it holds that $I^\Phi \models \psi\{\bar{x}/\bar{t}\}$ for all $\psi \in \Psi$. As ψ is atomic it follows from the definition of I^Φ that $\Phi \models \psi\{\bar{x}/\bar{t}\}$. As this is true for every $\psi \in \Psi$ it holds that $\Phi \models \bigwedge \Psi\{\bar{x}/\bar{t}\}$ for some sequence of terms $\bar{t}$. □

This proposition is important for logic programming for two reasons: (1) to decide whether a formula of the form $\exists \bar{x}. \bigwedge \Psi$ is entailed by a logic program can be decided simply by checking whether $\exists \bar{x}. \bigwedge \Psi$ is true in I^Φ (without extending Φ). And (2) if such an entailment exists, then there are terms which may be substituted for the variables in $\bar{x}$.

Therefore the logic programming problem can be decided by constructing a model for a set of universal Horn clauses. We now argue why, in the propositional case, models of horn clauses may be generated efficiently. We first consider the easier question of whether a model for a propositional formula exists.

4.2 Semantics of Logic Programs

4.2.1 Model Theoretic Semantics

Let us now turn from the logic programming problem to the problem of defining a semantics of a logic program. As we just discussed, Herbrand models of logic programs can be constructed efficiently. So why

not define the semantics of a logic program P to be the set of Herbrand models of P. Unfortunately even simple logic programs have an infinite number of Herbrand models

Example 4.1. Consider the logic program P

$$p(f(a))$$

The set of Herbrand models of P is

$$\{\{p(f(a))\}, \{p(f(a)), p(f(f(a)))\}, \{p(f(a)), p(f(f(f(f(a))))))\}, \ldots\}$$

Fortunately, it holds that the intersection of two Herbrand models of a logic program P is also a Herbrand model of P (proved in [55] 2.6.1). As each logic program has a Herbrand model (e.g B_P) the set of all Herbrand models is not empty. Therefore there is a models which is the intersection of all Herbrand models of a logic program. This model is called the *minimal Herbrand model* of P (denoted M_P). Usually the semantics of an logic program P is defined to be the least Herbrand model of P.

In this thesis we propose the query language RDFLog, which is a query language for RDF. We view RDF as a logical formula. As we want RDFLog to be closed, i.e. the output of the query language is in the same formalism as the input, we define the semantics of RDFLog to be a logic formula. To show that the semantics of RDFLog resembles the semantics of logic programming, we define the semantics of logic programming to be a logical formula.

Definition 22 (Semantics of Logic Programming)**.** We define the *semantics* $[\![P]\!]_l$ of a logic program P to be

$$[\![P]\!]_l := \bigwedge M_P$$

Observe that, as the least Herbrand model of a logic program P may be infinite, the semantics of a logic program may be an infinite conjunction of atoms.

4.2.2 Fixed Point Semantics

In this section we finally show how Herbrand models of sets of universal Horn clauses may be constructed efficiently. First consider the problem of finding a model for a set of propositional formulas (formulas, in which every predicate symbol is 0-ary). It is easy to see that models for propositional formulas in disjunctive normal form may be constructed in polynomial time, while it is still open whether this is possible for formulas in conjunctive normal form. In fact, finding a model of a propositional formula in conjunctive normal form in polynomial time amount to solving the famous $P = NP$ in a positive way!

As the models of a set of formulas Φ is equal to the models of $\bigwedge \Phi$, we are faced with a similar problem here. Therefore even propositional logic programming seems not to be tractable. But again, the restriction to Horn clauses helps. Deciding satisfiability for a set of propositional Horn clauses is in P:

- Let M be the empty Herbrand model.
- While there is a rule $a \leftarrow a_1, \ldots, a_n$ in P such that $a_1, \ldots, a_n$ are true in M add a to M.

It is easy to see that a fixed point is reached after $O(|P|)$ iterations.

The construction of models for non-propositional logic programs follows the same idea. We proceed as follows: We first define an operator for the general case of logic programming. Then we show that the minimal Herbrand model of a logic program P is equal to the least fixed point of this operator. Finally we show that this fixed point is reached in a countable number of iterations.

Definition 23 (Consequence Operator). Let P be a logic program and M a set of ground atoms. Let $r = a \leftarrow a_1, \ldots, a_n$ be a horn clause. Define $cons_r : 2^{B_A} \to 2^{B_A}$ as

$$cons_r(M) := \begin{cases} M \cup a & \text{if } \{a_1, \ldots, a_n\} \subseteq M \\ M & \text{otherwise} \end{cases}$$

The *immediate consequence operator* $T_P : 2^{U_A} \to 2^{U_A}$ is defined as

$$T_P(M) = \bigcup_{r \in gnd(P)} cons_r(M)$$

A *fixed point* of an operator $F : M \to M$ is an element $m \in M$ such that $F(m) = m$. If M is partially ordered, say by some $\prec$, then a fixed point m of F is the *least fixed point* of F if for all fixed points m' of F it holds that $m \prec m'$. Observe that Herbrand models may be partially ordered wrt. the subset relation.

Models of a logic program P can be constructed by iterating the consequence operator of P beginning from the empty set. As the least fixed point may not be reached after a finite number of steps, we define the iteration of the consequence operator for ordinal numbers.

$$\begin{aligned} T_P^0 &= \varnothing \\ T_P^\alpha &= T_P(T_P^{\alpha-1}) && \text{if } n \text{ is a successor ordinal} \\ T_P^\alpha &= \bigcup \{ T_P^\beta \mid \beta < \alpha \} && \text{if } n \text{ is a limit ordinal} \end{aligned}$$

Van Emden and Kowalski showed that the minimal Herbrand model of a logic program is equal the least fixed point of T_P. The proof uses a theorem due to Tarski. We do not give the proof here since it can be found e.g. in [55]. In addition it can be shown that T_P always has a fixed point and that this fixed point is reached after a countable number of iterations. Therefore we can state

Proposition 5. Let P be a logic program. Then $M_P = T_P^\omega$.

This result shows that this fixed point iteration is useful as a computational procedure to compute models of logic programs. The following corrollary is also easy to show by transfinite induction over the number of iterations of the operator:

Corollary 4.2. Let P be a logic program. Then it holds that

$$P \models [\![P]\!]_l$$

4.3 Backward Chaining

4.3.1 Why Backwards?

We discussed in the previous section that the semantics of a logic program might be infinite. Consider the logic program

Listing 1. Parents are Humans

```
human(hans) ← .
human(Y) ← is_child_of(X, Y).
is_child_of(X,parent(X)) ← human(X) .
```

It is easy to see that the minimal Herbrand model of this program is infinite. If we are only interested in whether Hans' father is human it is certainly not necessary to compute the whole model.

A different possibility to answer the question 'Is Hans' parent human?' would be to assume that the answer is 'no'. Then it follows from the second clause that nobody is a child of Hans' parent. In particular Hans is not a child of Hans' parent. Therefore by the last clause it holds that Hans is hot human which is a contradiction to the first clause. Therefore our first assumption must have been wrong and thus the answer to the above question is 'yes'.

So what is the benefit so such sorts of arguments? Instead of first enumerating everything that enforced to be true by a logic program we assumed the query to be wrong and found a refutation proof. This is exactly the idea of SLD-resolution: Given a set of universal positive Horn clauses Φ and a universal negative Horn clause $\forall \bar{x}(\bot \leftarrow a_1, \ldots, a_n) = \neg\exists \bar{x} \,.\, a_1, \ldots, a_n$, show that $\Phi \cup \{\neg\exists \bar{x} \,.\, a_1, \ldots, a_n\}$ is not satisfiable. Observe that $\Phi \cup \{\neg\exists \bar{x} \,.\, a_1, \ldots, a_n\}$ is unsatisfiable iff $\Phi \models \exists \bar{x} \,.\, a_1, \ldots, a_n$. By Proposition 4 it holds that there is a sequence $\bar{t}$ of terms such that $\Phi \models a_1, \ldots, a_n\{\bar{x}/\bar{t}\}$.

Reconsider our example argumentation. In one step we argued that as nobody is Hans' parents' child in particular Hans himself is not. Cast in logical terms this amounts to binding a term (Hans) to a universal variable (somebody). In SLD-resolution, the bodies of rules are matched against the heads of rules. The matching consists in finding a substitution σ such that if σ is applied to tow rules r_1, r_2, then the head of $r_1\sigma$ is equal to some body atom in $r_2\sigma$. It is desirable to compute most general unifiers in order to obtain as few answers as possible.

Definition 24 (Unifier)**.** Let M be a set of atoms. A substitution σ is called a *unifier* if $M\sigma$ is singleton. A unifier σ is called *most general* (denoted *mgu*) for a set of atoms M if for all unifiers τ of M it holds that there is a substitution ρ such that $\sigma\rho = \tau$.

The concept of unification goes back to Herbrand [43] in 1930. It was reconsidered in 1963 by Robinson [69] to reduce the size of the search space. Even though the size of a unification is exponential in general, there are algorithms which compute unifications in linear time [59, 58]. This is done by efficiently representing the unificator.

4.3.2 SLD Resolution

A new assumption in the above refutation proofs is called a SLD resolvent.

Definition 25 (SLD Resolvent)**.** Let $c = b \leftarrow b_1, \ldots, b_k$ be the clause, let $q = \leftarrow a_1, \ldots, a_m, \ldots, a_n$ a query, and let θ be the mgu of $\{a_m, b\}$. We assume that q and c have no variables in common (otherwise we rename the variables of c). Then q' is an *SLD resolvent of q and c using θ* if q' is the query $\leftarrow (a_1, \ldots a_{m-1}, b_1, \ldots b_k, a_{m+1}, \ldots a_n)\theta$.

A refutation proof is called a SLD derivation. Observe that there might be several such proofs.

Definition 26 (SLD Derivation). A *SLD derivation* of $P \cup \{q\}$ consists of a sequence $q_0, q_1, \ldots$ of queries where $q = q_0$, a sequence $c_1, c_2, \ldots$ of variants of program clauses of P and a sequence $\theta_1, \theta_2, \ldots$ of mgu's such that q_{i+1} is a resolvent from q_i and c_{i+1} using θ_{i+1}. An *SLD-refutation* is a finite SLD-derivation which has the empty query as its last query.

Given a logic program P and a query q, the set of all possible refutation proofs may be represented as a tree.

Definition 27 (SLD Tree). An *SLD tree T w.r.t. a program P and a query q* is a labeled tree where every node of T is a query and the root of T is q and if q is a node in T then q has a child q' connected to q by an edge labeled (c, θ) iff q' is an SLD-resolvent of q and c using θ.

Let P be a definite program and q a definite query. A *computed answer* θ for $P \cup \{q\}$ is the substitution obtained by restricting the composition of $\theta_1, \ldots \theta_n$ to the variables occurring in q, where $\theta_1, \ldots \theta_n$ is the sequence of mgu's used in an SLD-refutation of $P \cup \{q\}$.

Observe that in each resolution step the selected literal a_m and the clause c are chosen non-deterministically. We call a function that maps to each query one of its atoms a *computation rule*. The following proposition shows that the result of the refutation is independent of the literal selected in each step of the refutation.

Proposition 6 (Independence of the Computation Rule). [55] Let P be a definite Program and q be a definite query. Suppose there is an SLD-refutation of $P \cup \{q\}$ with computed answer θ. Then, for any computation rule R, there exists an SLD-refutation of $P \cup \{q\}$ using the atom selected by R as selected atom in each step with computed answer θ' such that $q\theta$ is a variant of $q\theta'$.

The independence of the computation rule allows us to restrict the search space: As a refutation corresponds to a branch of in an SLD-tree, to find all computed answers we need to search all branches of the SLD-tree. The independence of the computation rule allows us to restrict our search to branches constructed using some (arbitrary) computation rule.

Example 4.3. Consider the logic program 2 with query $q = \leftarrow t(1, 2)$:

Listing 2. Transitive Closure

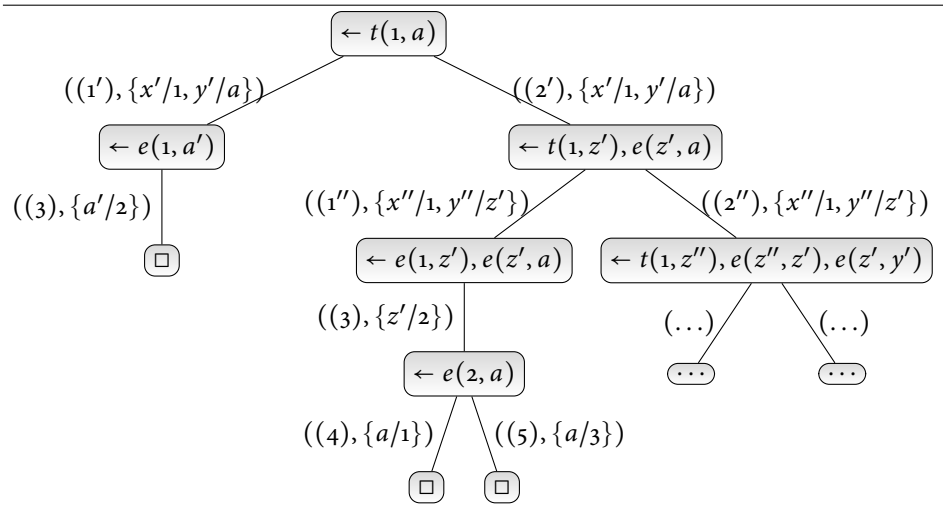
```
t(x,y) ← e(x,y).
t(x,y) ← t(x,z), e(z,y).
e(1,2) ← .
e(2,1) ← .
e(2,3) ← .
← t(1,2) .
```

An SLD-tree for program 2 and q is shown in Figure 3. The edges of the SLD tree are labeled with the number of a rule instead of a rule. We denote by (n') the rule obtained from rule n with each variable x is replaced by x'.

4.3.3 Soundness and Completeness of SLD Resolution

To compare the operational semantics of a program P to its declarative semantics we define a declarative counterpart of an answer to a query to a program P. A *correct answer* for a program P and query q is a

Figure 3 An SLD tree for program 2



substitution σ such that $[\![P]\!]_I \models q\sigma$. Using this notion we can define the soundness and completeness of logic programming.

Proposition 7 (Soundness and Completeness of Logic Programming). [55] Let P be a program and let q be a query. Then it holds that

- every computed answer of P and q is a correct answer and
- for every correct answer σ of P and q there exists a computed answer θ such that θ is more general that σ.

Observe that to find a computed answers of a program P and query q operationally one has to visit the leaf of a finite branch in the SLD-tree w.r.t. P and q. The order in which we visit these nodes is not determined by the definition of an SLD-refutation. We call such an order a *search strategy*. An *SLD-procedure* is a deterministic algorithm which is an SLD-resolution constrained by a computation rule and a search strategy.

As SLD-trees are infinite in general, the completeness of an SLD-procedure depends on the search strategy. To be complete, an SLD-procedure must visit every leaf of a finite branch of an SLD-tree within a finite number of steps. A search strategy with this property is called *fair*. Obviously not every search strategy is fair. For example the depth first search strategy used by Prolog is not fair. An example of a fair search strategy is breath first search.

4.4 Tabling

4.4.1 The Idea

As stated in the previous section not every search strategy is complete. This is due to the fact that an SLD-tree is infinite in general. As we only consider finite programs, an SLD-tree may only be infinite if it

has an infinite branch. As a branch in an SLD-tree corresponds to an SLD-derivation we denote a branch as $[q_1, q_2, \ldots]$ where $q_1, q_2 \ldots$ are the queries of the corresponding derivation.

A branch $B = [q_1, q_2, \ldots]$ in an SLD-tree may be infinite if there is a sub-sequence $[q_{i_1}, q_{i_2}, \ldots]$ ($i_j < i_k$ if $j < k$) of B such that

- for all $j, k \in \mathbb{N}$ q_{i_j} and q_{i_k} contain an equal (up to renaming of variables) atom or
- for all $j \in \mathbb{N}$ q_{i_j} contains an atom which is a real instance of an atom in $q_{i_{j+1}}$.

Non-termination due to the first condition is addressed by a evaluation technique called *tabling* or *memorization*. The idea of tabling is the idea of dynamic programming: store intermediate results to be able to look these results up instead of having to recompute them. In addition to the better termination properties, performance is improved with this approach.

4.4.2 The Algorithm

The OLDT algorithm [75] is an extension of the SLD-resolution with a left to right computation rule. Like SLD-resolution, it is defined as a non-deterministic algorithm.

A subset of the predicate symbols occurring in a program are classified as *table predicates*. A query is called a *table query* if its leftmost atom has a table predicate. Solutions to table queries are the intermediate results that are stored. Table queries are classified as either *solution queries* or *look-up queries*. The intuition is that a solution query 'produces' solutions while a look-up query looks up the solutions produced by an appropriate solution query.

An *OLDT-structure* (T, T_S, T_L) consists of an SLD-tree T and two tables, the solution table T_S and the look-up table T_L. The *solution table* T_S is a set of pairs $(a, T_S(a))$ where a is an atom and $T_S(a)$ is a list of instances of a called the *solutions* of a. The *look-up table* T_L is a set of pairs $(a, T_L(a))$ where a is an atom and p is a pointer pointing to an element of $T_S(a')$ where a is an instance of a'. T_L contains one pair $(a, T_L(a))$ for an atom a occurring as a leftmost atom of a query in T.

The *extension of an OLDT structure* (T, T_S, T_L) consists of three steps:

1. a resolution step,
2. a classification step, and
3. a table update step.

In the resolution step a new query is added to the OLDT-tree, in the classification step this new query is classified as either non-tabled query or solution query or look-up query and in the table update step the solution table and the update table are updated. While step one is equal for non-tabled and solution queries, step two and three are equal for tabled nodes while there is nothing to do in these steps for non-tabled nodes.

Let (T, T_S, T_L) be an OLDT structure and $q = \leftarrow a_1, \ldots, a_n$ a query in T. If q is a non-tabled query or a solution query then in the resolution step a new query q' is added to T which is connected to q with an edge labeled (c, θ) where q' is the SLD-resolvent of q and c using θ. If q is a look-up node then in the resolution step the new node q' is added to T with an edge labeled $(a \leftarrow, \theta)$ where a is the atom in the

solution table that the pointer $T_L(a_1)$ points to and the substitution θ is the mgu of a and a_1. Finally the pointer $T_L(a_1)$ is set to point to the next element of the list it points to.

In the classification step the new query q' is classified as a non-table query if its leftmost atom is not a table predicate and a table query otherwise. If q' is a table query then q' is classified as a look-up node if there is a pair $(a, T_S(a))$ in the solution table and a is more general than the leftmost atom a' of q'. In this case a new pair (a', p) is added to the look-up table and p points to the first element of $T_S(a)$. If q' is not classified as a look-up node then it is classified as a solution node and a new pair $(a', [])$ is added to the solution table.

In the table update step new solutions are added to the solution table. Recall that the problem we want to tackle here is the recurrent evaluation of equal (up to renaming of variables) atoms in queries. Therefore the 'solutions' we want to store in the solution table are answers to an atom in a query.

In SLD-resolution the term answer is defined only for queries. This notion can be extended to atoms in queries in the following way. OLDT-resolution uses a left to right computation rule. If the derivation of a query $q =\leftarrow a_1, \ldots, a_n$ is finite, then there is a finite number n of resolution steps such that the nth resolvent q_n on q is $\leftarrow a_2, \ldots, a_n$. We call the sequence $[q_1, \ldots, q_n]$ a *unit sub-refutation of* a_1 and the restriction of $\theta_1 \ldots \theta_n$ to the variables occurring in a_1 is called an *answer for* a_1.

Now if the query q produced in the resolution step is the last query of a unit sub-refutation of a with answer θ then the update step consists in adding θ to the list $T_S(a)$.

4.4.3 An Example

Example 4.4. Reconsider the program from Example 4.3

Listing 3. Transitive Closure

```
t(x,y) ← e(x,y) .
t(x,y) ← t(x,z), e(z,y) .
e(1,2) ← .
e(2,1) ← .
e(2,3) ← .
← t(1,2) .
```

After a sequence of OLDT-resolutions of solution queries or non-tabled queries the OLDT-tree in Figure 4 is constructed. To indicate which nodes are solution nodes and which are look-up nodes we prefix solution nodes with 'S:' and look-up nodes with 'L:'.

As the left branch is a unit sub-refutation of $t(1, a)$ with solution $\{a/2\}$ the entry $t(1, 2)$ is added to the solution table. As $t(1, a)$ is more general than the leftmost atom of the query $t(1, z'), e(z', a)$ this query is classified as a look-up node. Instead of using resolution to compute answers for the first atom of this query we use the solutions stored in the solution table. The final OLDT-tree is depicted in 5:

Observe that the program of example 4.4 does not terminate with SLD-resolution while it does terminate with OLDT-resolution. The following example shows that OLDT-resolution is not complete in general.

Example 4.5. Consider the program 4 and query $q = \leftarrow$ `p(x)`

Figure 4 An intermediary OLDT tree for program 3

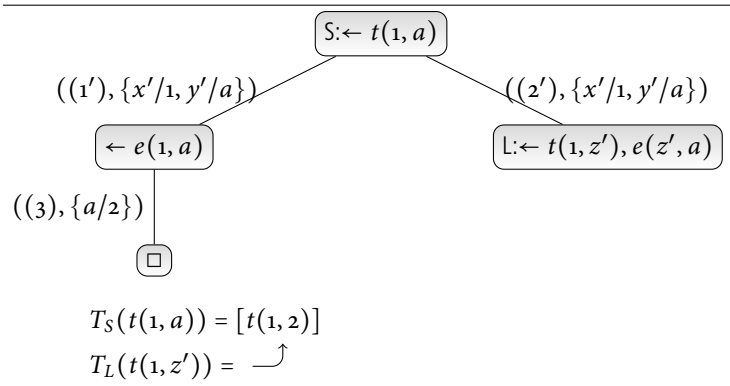
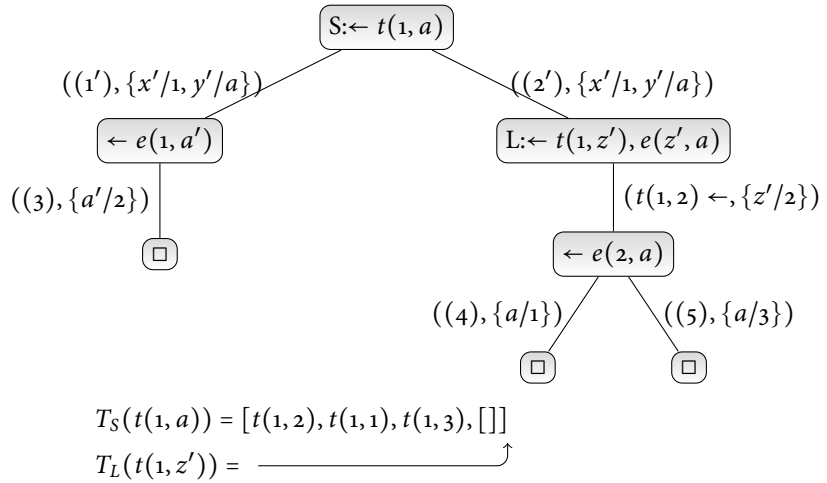


Figure 5 The final OLDT tree for program 3



Listing 4. Program for which OLDT resolution is incomplete

```
p(x) ← q(x), r .
q(s(x)) ← q(x) .
q(a) ← .
r ← .
← p(x) .
```

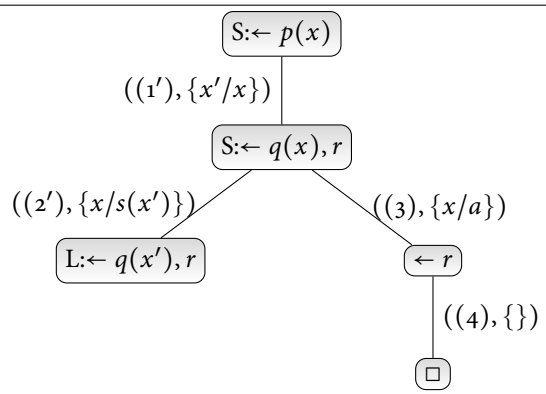
After a sequence of OLDT-resolution steps the OLDT-tree in Figure 6 is constructed

In the next step the solution $q(a)$ can be used to generate the solution $q(s(a))$ (see Figure 7).

It is easy to see that if reduction steps are only applied to the node L:← $q(x')$, r then no solutions for $p(x)$ will be produced in finite time. Therefore OLDT is not complete in general.

This problem was addressed by the authors of OLDT. They specified a search strategy called *multistage depth-first strategy* for which they showed that OLDT becomes complete if this search strategy is used. The

Figure 6 An intermediary OLDT tree for program 4



$T_S(p(x)) = []$

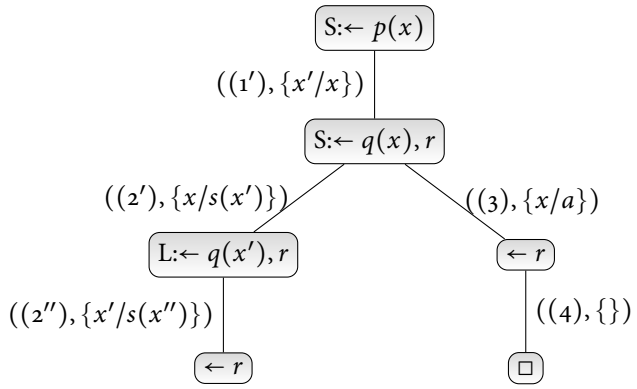
$T_S(q(x)) = [q(a)]$

$T_L(q(x')) = \curvearrowright$

idea of this search strategy is to order the nodes in the OLDT-tree and to apply OLDT-resolution-steps to the nodes in this order. If the node that is the biggest node with respect to that ordering is reduced then a stage is complete and a new stage starts where reduction is applied to the smallest node again. Therefore it is not possible to apply OLDT-steps twice in a row if there are other nodes in the tree which are resolvable.

In the above example it would therefore not be possible to repeatedly apply reductions to the node L:← $q(x'), r$ without reducing the node ← r which yields a solution for $p(x)$.

Figure 7 An intermediary OLDT tree for program 4



Part II

RDFLog

Chapter 5

Syntax and Examples

In this section we introduce the language RDFLog. We show using example how RDFLog can be used to query RDF graphs with blank nodes. In Section 5.2 we demonstrate the construction and grouping capabilities of RDFLog. Finally in Section 5.3 we define the syntax of RDFLog programs. The definition of RDFLog is deferred until the next Chapter.

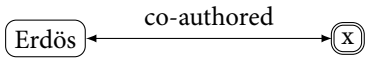
5.1 RDF querying with RDFLog

Recall the example RDF graph from Section 3.1 depicted in Figure 8.

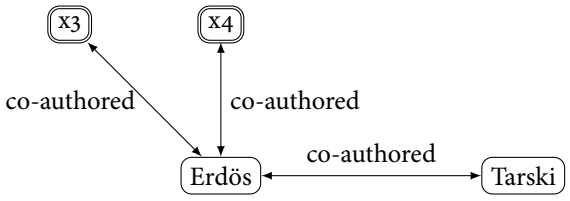
RDFLog allows to extract subgraphs from an RDF graph using an RDFLog query.

Definition 28 (RDFLog Query, Answer, Answer Graph)**.** An *RDFLog query* is an RDF graph, possibly containing blank nodes. Given an RDF graph g and an RDFLog query q, an *answer* of q and g is an RDF graph which is an instance of q and a subgraph of g. The *answer graph* for g and q is the union of the answers of g and q without renaming of blank nodes (simply called union in the RDF semantics [40]).

For example the consider the RDFLog query q asking for the direct co-authors of Erdös.

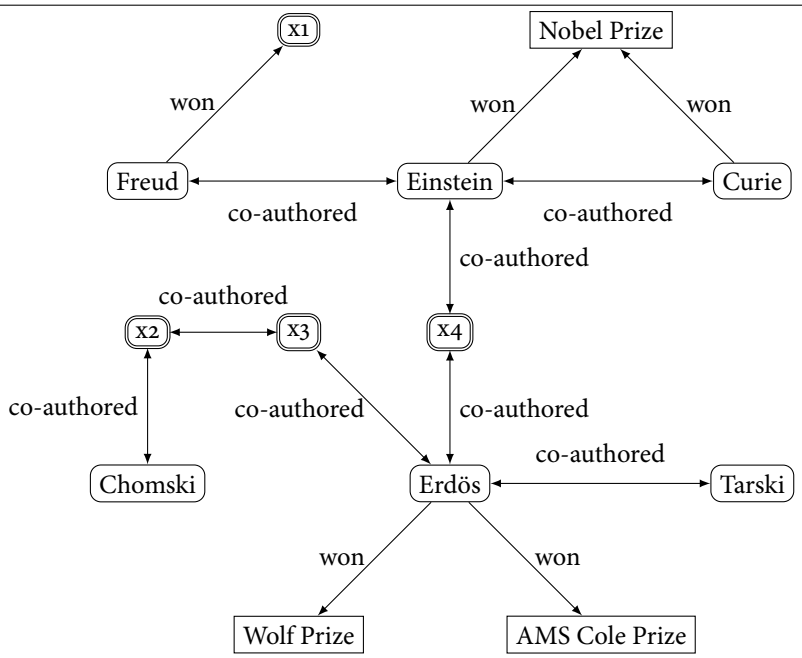


The answer graph for q is



Observe that even though the co-author graph is lean, the answer graph of q is not: it is equivalent to its subgraph consisting only of the nodes representing Erdös and Tarski. Nonetheless as the co-author graph

Figure 8 A co-author graph in RDF

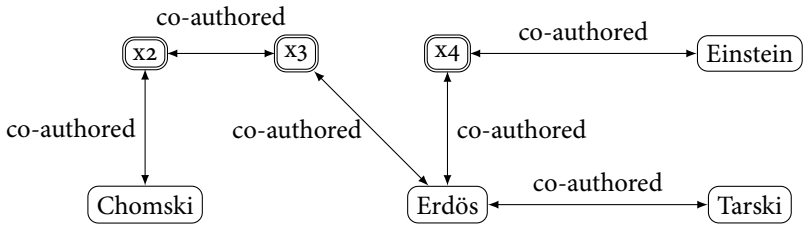


is lean it is guaranteed under a closed world assumption that the nodes labeled x3, x4 do not represent the same entity as the node labeled Tarski: While x4 co-authored with Einstein, Tarski did not. This shows that it might be useful to extend the notion of answers to concise bounded descriptions: whenever there is a blank node in an answer and there is an edge between this blank node and another node then this edge is added to the answer. Again the concise bounded answer graph is the union of all concise bounded answers.

Definition 29 (Concise Bounded Description, Concise Bounded Answer, Concise Bounded Answer Graph)**.** Let g be an RDF graph and b a blank node. The *concise bounded description* of b in g is the subgraph of g consisting of all nodes which may be reached from b or may reach b by a path containing only blank nodes.

The *concise bounded answer* of an RDFLog query q and g is the union of the answers for q and g and all concise bounded descriptions of blank nodes occurring in this answer. The *concise bounded answer graph* is the union of all concise bounded answers of q and g.

The concise bounded answer graph for the co-author graph and the afro-mentioned query is:



5.2 Construction with RDFLog

RDFLog can be seen as a rule extension of RDF. Cast in the language of deductive databases, an RDF graph represents an extensional databases, which may be augmented by RDFLog rules to define an intentional database. As an RDF triple can be modelled as an RDFLog rule with an empty body, an RDFLog program can be considered as a set of RDFLog rules.

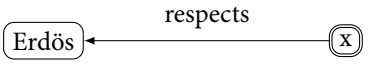
RDFLog rules are similar to Datalog rules. The difference is that in contrast to Datalog, existential variables may occur in the head. These existential variables are needed to construct new blank nodes. As both existential and universal variables may occur in a rule, it is necessary to explicitly quantify theses rules. The semantics of an RDFLog rule depends on the order of the quantifiers.

Instead of giving a definition of the intentional database defined by a set of RDFLog rules, in this section we provide an intuition and show some examples. The exact semantics is defined in Section 7. Given an RDF graph g and a set P of RDFLog rules, the head of a rule r in P is added to g if the body of r matches against g. A fresh blank node y is constructed for every binding of universal variables with quantifiers to the left of the quantifier of y. This specification mimics the semantics of quantifiers in both natural language and logics.

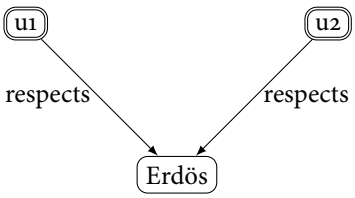
Consider for example the statement, that whenever someone has won a prize, then there is someone who respects him (for example the guy awarding the prize). In RDFLog this can be formulated by the following rule.

```
1 ∀ res prz ∃ res' . won(res,prz) →
  respects(res',res)
```

Now it is possible to pose queries against the intentional database. The following query selects all edges of the RDF graph in Figure 8 stating that someone respects Erdös.



The answer graph to the above query and the RDFLog program consisting of the above rule and the co-author graph is

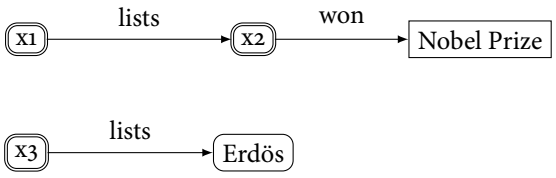


Observe that as Erdös won two prizes there, RDFLog creates two new blank nodes. This is precisely how the above rule reads: for every pair of researcher and prize there is someone who respects the prize winner. It should be noted that blank nodes in the head of an RDFLog rule have to be renamed, as a rule may be used more than once.

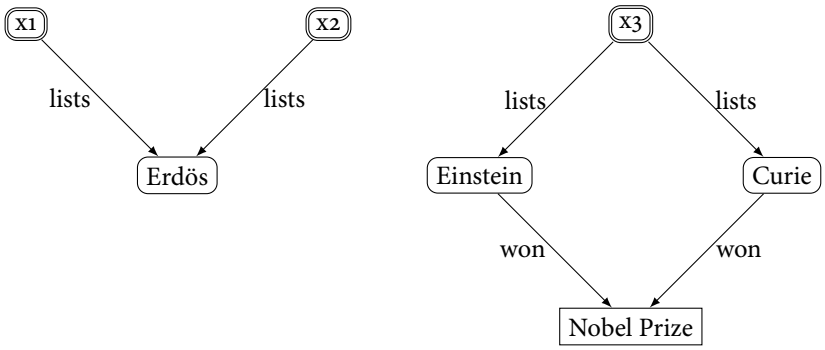
Assume we want to group researchers by prize. For example one could state that for every prize, there is a web page listing all the prize winners. This could be formulated in RDFLog by the following rule:

```
∀ prz ∃ page ∀ res . won(res,prz) → lists(page,res)
```

We can now query for all listings of Erdös and a Nobel prize winners with the following query.



The answer graph for this query is



It is easy to see that the researchers are grouped by the prizes they one.

5.3 Syntax of RDFLog

In this section we define the syntax of RDFLog. The syntax we use is a restricted form of first order logic. The reason for this is that the semantics of RDFLog is inspired from logics. Formulas are not suited as a programming language for various reasons. It would therefore be tempting to develop a more user friendly syntax for RDFLog. As the topic of this deliverable is to investigate the properties of the semantics of RDFLog, this is left as future work.

Definition 30 (RDFLog Rule, RDFLog Scope, RDFLog Program)**.** An *RDFLog rule* is a formula of the form

$$a_1 \wedge \ldots \wedge a_n \rightarrow b_1 \wedge \ldots \wedge b_m$$

An *RDFLog scope* is a formula of the form

$$\forall \bar{x}_1 \exists \bar{y}_1 \ldots \forall \bar{x}_k \exists \bar{y}_k \,.\, r_1 \wedge \ldots \wedge r_l$$

where $r_1, \ldots, r_l$ are RDFLog rules. Observe that n, m, l, k and the sequences ocurring in the definition are finite. An *RDFLog program* is a finite conjunction of RDFLog scopes.

We often ommit the conjunction symbol between rules and denote the conjunction symbol within a rule by ','. Observe that an RDFLog program is a *finite* formula.

Chapter 6

Evaluating RDFLog

6.1 Skolemisation and Back

In this section we introduce the Skolemisations and the un-Skolemisation. Theses are our main tools for the reduction of RDFLog programs to logic programs. The Skolemisation is a well known mapping from formulas to universal formulas. We need to adapt the definition of Skolemisation to the infinite formulas defined in Section 2. We show that this extention has the same properties as the well known definition for finite formulas. We then define a new notion of un-Skolemisation, again for infinite formulas. We show that the our definition of un-Skolemisation enjoys properties symmetric to those of Skolemisation.

6.1.1 Skolemisation

A Skolemisation is a function that maps a formula φ to a formula ψ such that

- $\psi \models \varphi$ and
- if φ is satisfiable then so is ψ and
- ψ contains no existential variables.

In this section we define the Skolemisation and its inverse function, the un-Skolemisation. We show that the un-Skolemisation has properties symmetric to those of the Skolemisation.

Example 6.1. We use the following *RDFLog* program P as a running example throughout this section:

$$\exists y_1 \forall x_1 x_2 \exists y_2 \,.\, p(a, y_1) \wedge p(x_1, x_2) \to q(x_2, y_2)$$

A Skolemisation replaces each existential variable y in a formula φ by a term t such that t does unify with any other term in φ and t contains exactly the universal variables preceding y in the prefix of φ. As we can consider *RDFLog* programs formulas we can apply Skolemisation to *RDFLog* programs. For example if s is a Skolemisation which replaces y_1 by the constant c and y_2 by the term $f(x_1, x_2)$, then the application of s to P yields

$$\forall x_1 x_2 \,.\, p(a, c) \wedge p(x_1, x_2) \to q(x_2, f(x_1, x_2))$$

The minimal Herbrand model of this logic program is

$$\{p(a,c), q(c, f(a,c))\}.$$

We now define the Skolemisation. To avoid confusion recall that we defined the function symbol of a term $f(t_1, \ldots, t_n)$ to be f and that sequences may be empty.

Definition 31 (Skolemisation)**.** A *Skolemisation* s is a mapping from formulas to formulas. It is defined by a sequence $[s_1, \ldots, s_n]$ of mappings from variables to terms. The set of function symbols of terms in the range of $s_1 \cup \ldots \cup s_n$ is denoted by S.

The empty Skolemisation $[\,]$ is applicable to any formula. A Skolemisation $s = [s_1, \ldots, s_n]$ is *applicable* to a formula $\varphi = \forall \bar{x} \exists \bar{y} \psi$ over alphabet A if (1) A and S are disjoint, (2) if $t, t' \in ran(s_1 \cup \ldots \cup s_n)$ and $t \neq t'$ then t and t' do not unify, (3) the domain of s_1 is a superset of $\bar{y}$, (4) all terms in the range of s_1 contain exactly the variables $\bar{x}$ and (5) $[s_2, \ldots, s_n]$ is applicable to $[s_1](\varphi)$.

The application of a skolemisation $s = [s_1, \ldots, s_n]$ to a formula $\varphi = \forall \bar{x} \exists \bar{y} \psi$ is defined as

$$s(\varphi) := \begin{cases} \varphi & \text{if } s = [\,] \\ s'(\forall \bar{x}.\psi\{\bar{y}/s_1(\bar{y})\}) & \text{otherwise} \end{cases}$$

where $s' = [s_2, \ldots, s_n]$. The set of terms in the domain of a skolemisation is called *Skolem terms*.

Example 6.2. Reconsider the Skolemisation from Example 6.1. We can now formalise this Skolemisation as $s = [s_1, s_2]$ where $s_1(y_1) = c$ and $s_2(y_2) = f(x_1, x_2)$.

The definition is a generalisation of the usual definition in that usually each existential variable is replaced by a term with a unique function symbol. We use this more general form where it is only required that terms in the domain of a Skolemisation do not unify with each other for reasons that will become clear later. We first show that if terms in the range of s do unify then satisfiability is no preserved over Skolemisation anymore. Consider the formula

$$\exists x \forall y \exists z. \neg p(a,x) \wedge p(y,z)$$

This formula is satisfiable for example by an interpretation I with interprets p by $\{(a,b),(b,b)\}$. Now assume we replace x by $f(a)$ and z by $f(y)$. We obtain the formula

$$\forall y. \neg p(a, f(a)) \wedge p(y, f(y))$$

which is not satisfiable anymore.

One is often interested in a Skolemisation that replaces *all* existential variables in a given formula. This is captured by the notion of an induced Skolemisation:

Definition 32 (Induced Skolemisation)**.** A Skolemisation which eliminates all existentially quantified variables in a formula φ is called an *induced Skolemisation* of φ.

6.1.2 un-Skolemisation

We define the un-Skolemisation such that it has the following properties: If φ is a formula and ψ is the un-Skolemisation of φ then it holds that:

- $\varphi \models \psi$.
- If ψ is satisfiable then so is φ.
- The un-Skolemisation is the inverse of the Skolemisation and vice versa.

Observe that given a set of function symbols S and a first order formula φ it is not always possible to return a formula ψ such that φ is the Skolemisation of ψ and such that φ is the Skolemisation of ψ (i.e. φ is the un-Skolemisation of φ by the third requirement). Consider for example the formula

$$\forall x_1 x_2 \,.\, p(x_1, x_2, f(x_1), g(x_2))$$

It is not possible to replace all terms with function symbols f or g such that our requirement are fulfilled. The formula states the value of the third argument of p depends only and the value of the first argument depends only on the second. Henkin introduced the so called *Henkin quantifiers* [42] such that the following is a formula.

$$\begin{pmatrix} \forall x_1 & \exists y_1 \\ \forall x_2 & \exists y_2 \end{pmatrix} . p(x_1, x_2, y_1, y_2)$$

The semantics of this formula is exactly the semantics we are looking for. It was shown by Ehrenfeucht that first order logic with Henkin quantifiers are not expressible in first order logic, by showing that the quantifier 'for infinitely many x' is expressible in such a language. Blass and Gurevich showed that this logic captures existential second order logic [14]. Badia and Vansummern have recently showed how to restrict Henkin quantifiers in such a way that expressive power does not extend first order logic [7]. For our purposes it is sufficient to restrict the formulas to which un-Skolemisation is applicable in such a way that the result is alway a first order formula.

Definition 33 (Un-Skolemisation)**.** An *un-Skolemisation* u is a mapping from formulas to formulas. It is defined by a sequence $[u_1, \ldots, u_n]$ of mappings from terms to variables. The set of function symbols of terms in the domain of $u_1 \cup \ldots \cup u_n$ is denoted by U.

The empty un-Skolemisation $[\,]$ is applicable to any formula. An un-Skolemi-sation $[u_1, \ldots, u_n]$ is *applicable* to a formula $\varphi = \forall \bar{x} \psi$ over alphabet A if (1) A and U are disjoint (2) if $t, t' \in dom(s_1 \cup \ldots \cup s_n)$ then t and t' do not unify, (3) in all terms in the domain of u_1 contain exactly the variables $\bar{x}$, (4) $[u_2, \ldots, u_n]$ is applicable to $[u_1](\varphi)$.

The application of an un-Skolemisation $u = [u_1, \ldots, u_n]$ to a formula $\varphi = \forall \bar{x} \psi$ is defined as

$$u(\varphi) := \begin{cases} \varphi & \text{if } s = [\,] \\ u'(\forall \bar{x} \exists \bar{y} . \psi\{\bar{t}/u_1(\bar{t})\}) & \text{otherwise} \end{cases}$$

where $\bar{t} = dom(u_1)$ and $u' = [u_2, \ldots, u_n]$.

Observe that the same example as above can be used to show that satisfiability is not preserved if terms in the domain of un-Skolemisations may unify with each other.

Example 6.3. Recall our Skolemised *RDFLog* program $s(P)$ from Example 6.1

$$\forall x_1 x_2 \,.\, p(a,c) \wedge p(x_1,x_2) \rightarrow q(x_2, f(x_1,x_2))$$

If we apply the un-Skolemisation $u = [u_1, u_2]$ with $u_1(f(x_1,x_2)) = y_2$ and $u_2(c) = y_1$ to $s(\varphi)$ we obtain the formula $u \circ s(P)$:

$$\exists y_1 \forall x_1 x_2 \exists y_2 \,.\, p(a,y_1) \wedge p(x_1,x_2) \rightarrow q(x_2,y_2)$$

Observe that in this case $s \circ u(P) = P$.

6.1.3 Properties of Skolemisation and un-Skolemisation

We show that it holds on general that if we are given a Skolemisation s, we can define an un-Skolemisation u such that $s \circ u(\varphi) = \varphi$ for any formula φ. We first show how we can obtain u from s and then show that u is the inverse of s.

Definition 34 (Un-Skolemisation of a Skolemisation)**.** Let $s = [s_1, \ldots, s_n]$ be a Skolemisation. Then the *un-Skolemisation of* s (denoted s^{-1}) is the un-Skolemisation $[s_n^{-1}, \ldots s_1^{-1}]$.

Let $u = [u_1, \ldots, u_n]$ be a un-Skolemisation. Then the *Skolemisation of* u for φ (denoted s^{-1}) is the Skolemisation defined by $[u_n^{-1}, \ldots u_1^{-1}]$ if this is a Skolemisation.

Lemma 5 (Un-Skolemisation is the Inverse of Skolemisation). (a) Let s be a Skolemisation and $u = s^{-1}$ its un Skolemisation. Then for all formulas φ to which s is applicable it holds that

$$u \circ s(\varphi) \models\!\models \varphi.$$

(b) Symmetrically, if u is an un-Skolemisation and $s = u^{-1}$ its un-Skolemisation. Then for all formulas φ to which u is applicable it holds that

$$s \circ u(\varphi) \models\!\models \varphi.$$

Proof. (a) Let $s = [s_1, \ldots, s_n]$ and $u = [s_m^{-1}, \ldots, s_1^{-1}]$. We show by induction on the length m of s that $[s_m^{-1}, \ldots, s_1^{-1}] \circ s(\varphi) = \varphi$. If s is of length o the statement is trivial.

For the inductive step we first show that this claim holds if s is defined by a sequence of length one. Thus, let $s = [s_1]$, and $s_1 = \{y_1 \mapsto t_1, y_2 \mapsto t_2, \ldots\}$ and let φ be a formula to which s is applicable. Therefore φ is of the form $\forall \bar{x} \exists \bar{y}.\psi$ where ψ does not start with an existential quantifier and $\bar{y} = [y_1, y_2, \ldots]$. The application of s to φ is

$$s(\varphi) = \forall \bar{x}.\psi\{y_1/t_1, y_2/t_1, \ldots\} \qquad (\star)$$

It is easy to see that $[s_1^{-1}] = [\{t_1 \mapsto y_1, t_2 \mapsto y_2, \ldots\}]$ applicable to $(\star)$ and by definition its application yields

$$\forall \bar{x} \exists \bar{y}.\psi\{y_1/t_1, y_2/t_1, \ldots\}\{t_1/z_1, t_2/z_1, \ldots\} \qquad (\star\star)$$

It is easy to show that $(\star\star) = \varphi$. Therefore $u(s(\varphi)) \models\!\!\mid \varphi$ if s is defined by a sequence of length one.

Let n be any fixed natural number. We now prove that

$$[s_n^{-1}, \ldots, s_i^{-1}] \circ [s_1, \ldots, s_n](\varphi) \models\!\!\mid [s_1, \ldots, s_{i-1}](\varphi)$$

by induction over $k = (n - i) + 1$. Observe that as we count downwards it holds that $[s_n^{-1}, \ldots, s_{n+1}^{-1}] = [\,]$. Then statement (a) follows for $i = 1$.

- $k = 0$: Thus i must be $n + 1$. As $[s_n^{-1}, \ldots s_{n+1}^{-1}] = [\,]$, this case is trivial.
- $k \to k + 1$: Therefore we need to show $i \to i - 1$.

$$\begin{aligned}
&[s_n^{-1}, \ldots, s_{i-1}^{-1}] \circ [s_1, \ldots, s_n](\varphi) \\
&= [s_{i-1}^{-1}] \circ [s_n^{-1}, \ldots, s_i^{-1}] \circ [s_1, \ldots, s_n](\varphi) \\
&\models\!\!\mid [s_{i-1}^{-1}] \circ [s_1, \ldots, s_{i-1}](\varphi) && \text{by hypothesis} \\
&= [s_{i-1}^{-1}] \circ [s_{i-1}] \circ [s_1, \ldots, s_{i-2}](\varphi) \\
&\models\!\!\mid [s_1, \ldots, s_{i-2}](\varphi) && \text{because } |[s_{i-1}]| = 1
\end{aligned}$$

(b) analog

□

We are now able to prove the main result of this section which is that the Skolemisation preserves satisfiability but models are preserved only in one direction. As we already noted the un-Skolemisation has symmetric properties.

Proposition 8 (Skolemisation Lemma). Let φ be a formula over alphabet $A \cup U$ in prenex normal form such that the prefix of φ is finite. Let S be a set of function symbols such that A, U and S are disjoint. Let s be the induced Skolemisation on S of φ and u an un-Skolemisation on U which is applicable to φ. Then it holds that

(a) $s(\varphi) \models \varphi$.

(b) $\varphi \models u(\varphi)$

(c) $s(\varphi)$ is satisfiable iff φ is satisfiable.

(d) φ is satisfiable iff $u(\varphi)$ is satisfiable.

Before we prove the Lemma, we first show that the proposition is true for our running example.

Example 6.4. Consider the entailment of $s(P)$

$$\begin{aligned}
\varphi = \forall u_1 u_2 \,.\, & p(a, c) \wedge p(b, u_2) \\
& q(c, f(a, c)) \wedge q(u_1, f(b, u_1)).
\end{aligned}$$

An un-Skolemisation which is applicable to φ is

$$u = [\{f(b, u_1) \mapsto v_3\}, \{c \mapsto v_1, f(a, c) \mapsto v_2\}]$$

We show that the statement (b) is true for φ by showing that

$$u(\varphi) = \exists v_1 v_2 \forall u_1 \exists v_3 \forall u_2 \,.\, p(a, v_1) \wedge p(b, u_2) \\ q(v_1, v_2) \wedge q(u_1, v_3).$$

is entailed by φ.

Therefore assume that $I = (D, i, \beta)$ is a model of φ. As I is a model of φ it holds that for all $\beta : \{u_1\} \to D$ it holds that $I\frac{\beta(u_1)}{u_1} \models \psi$ where $\psi = \forall u_2 . p(a, c) \wedge \ldots \wedge q(u_1, f(b, u_1))$. Let $d \in D$ be such that $I\frac{\beta(u_1)}{u_1}(f(b, u_1)) = d$. By the substitution Lemma we can replace the term $f(b, u_1)$ in ψ by a fresh variables v_3 if we map v_3 to d in I. Stated formally: For all $\beta : \{u_1\} \to D$ there is a $d \in D$ such that

$$I\frac{\beta(u_1)}{u_1}\frac{d}{v_3} \models \psi\{f(b, u_2)/v_3\}$$

By definition it holds that this is equivalent to

$$I \models \forall u_1 \exists v_3 \forall u_2 .\; p(a, c) \wedge p(b, u_2) \\ q(c, f(a, c)) \wedge q(u_1, v_3)$$

which is $[\{f(b, u_1) \mapsto v_3\}](\varphi)$. A similar argument can be made for the application of $[\{c \mapsto v_1, f(a, c) \mapsto v_2\}]$ to $[\{f(b, u_1) \mapsto v_3\}](\varphi)$.

Proof. **of Proposition 8** We first show (b) as (a) follows from (b) together with Lemma 5.

(b) Let I be a model of φ and let $u = [u_1, \ldots, u_1]$. We show the statement by induction on the length k of the prefix $[u_1, \ldots, u_k]$ of u, which has already been applied to φ

- $k = 0$: trivial.
- $k \to k+1$: Assume that I is a model of $[u_1, \ldots, u_k](\varphi)$ and that $u_{k+1} = \{t_1 \mapsto y_1, \ldots, t_n \mapsto y_n\}$. As u is applicable to φ it holds that φ is of the form $\forall \bar{x}.\psi$ and all the t_i with $1 \leq i \leq n$ contain exactly the variables $\bar{x}$. Therefore for all $\beta : \bar{x} \to D$ it holds that

  $$I\frac{\beta(\bar{x})}{\bar{x}} \models \psi$$

  Let $\bar{t} = [t_1, \ldots, t_n]$. Then there is a sequence $\bar{d} \in D^n$ such that $I\frac{\beta(\bar{x})}{\bar{x}}(\bar{t}) = \bar{d}$. It follows from the substitution lemma that

  $$I\frac{\beta(\bar{x})}{\bar{x}}\frac{\bar{d}}{\bar{y}} \models \psi\{\bar{t}/\bar{y}\}$$

  and therefore that

  $$I \models \forall \bar{x} \exists \bar{y} \psi\{\bar{t}/\bar{y}\}.$$

  The claim follows from the fact that $\forall \bar{x} \exists \bar{y} \psi\{\bar{t}/\bar{y}\} \models\!\mid [u_1, \ldots, u_{k+1}](\varphi)$.

(a) Let I be a model of $s(\varphi)$ and let $u' = s^{-1}$. It follwos from (b) that I is a model of $u'(s(\varphi))$. By Lemma 5 it holds that $u'(s(\varphi)) \models\!\mid \varphi$. Therefore I is a model of φ.

□

Again, before showing (c) and (d) we consider our Examples.

Proof. **of Proposition 8**

(c) The direction from left to right is immediate from (a). Let I be a model of φ. We first define an interpretation I_s which is an extension of I on S and then show by induction that I_s is a model of $s(\varphi)$.

Let T be the set of terms with a function symbol in S in the range of $s_1 \cup \ldots \cup s_n$. For each $t = f(t_1, \ldots, t_m) \in T$ we construct a function F_t in the following way:

Let y be the variable with $s(y) = t$. It holds that φ is of the form

$$\forall \bar{x}_1 \exists \bar{y}_1 \ldots \forall \bar{x}_i \exists \bar{y}_i \exists y.\psi$$

for some $i \geq 1$ where $\bar{y}_i$ might be empty. As $I \models \varphi$ it therefore holds that for all $\beta_{\bar{x}_1}$ there is a $\beta_{\bar{y}_1} \ldots$ there is a $\beta_{\bar{y}_i}$ and a β_y such that

$$I\frac{\beta_{\bar{x}_1}}{\bar{x}_1}\ldots\frac{\beta_{\bar{y}_i}}{\bar{y}_i}\frac{\beta_y}{y} \models \psi \qquad (\star)$$

Let B be the set of all sequences $b = [\beta_{\bar{x}_1}, \ldots, \beta_{\bar{y}_i}]$ for which $(\star)$ is true and denote by I_b the interpretation $I\frac{\beta_{\bar{x}_1}(\bar{x}_1)}{\bar{x}_1}\ldots\frac{\beta_{\bar{y}_i}(\bar{y}_i)}{\bar{y}_i}$. Observe that by the definition of the Skolemisation, t contains exactly the variables $\bigcup \bar{x}_1, \ldots, \bar{x}_i$. As φ has a finite prefix it holds that this sequence is finite. Therefore we can define $F_t : D^m \to D$ as

$$F_t(I_b(t_1), \ldots, I_b(t_n)) = \beta_y(y) \quad \text{iff} \quad b \in B.$$

Observe that as it holds by the definition of the Skolemisation that no two Skolem terms unify. In addition as all Skolem terms have a function symbol from U and A and U are disjoint it holds that a Skolem term unifies with no other term in φ. Therefore the function F_t is well defined.

We define $F' = \bigcup_{t \in T} F_t$ and let F'' be a function mapping any $\bar{d}$ which is not in the domain of F' some $d \in D$. Finally let $F = F' \cup F''$. In this way we obtain a function F for every function symbol $f \in sym(s)$.

If $I = (D, i, \beta)$, then we define $I'_s = (D, I \cup i', \beta)$ where

$$f^{i'} = F$$

We now show the other by induction over the size of $sym(s) = \{f_1, \ldots, f_k\}$ that I_s is a model of $s(\varphi)$.

- $k = 0$: In this case s must be the empty Skolemisation $[]$ and therefore the statement is trivial.

– $k-1 \to k$: As I is a model of $\varphi = \forall \bar{x}_1 \exists \bar{y}_1 . \psi$ it holds that for all $\beta_{\bar{x}_1} : \bar{x}_1 \to D$ there is a $\beta_{\bar{y}_1} : \bar{y}_1 \to D$ such that

$$I \frac{\beta_{\bar{x}_1}(\bar{x}_1)}{\bar{x}} \frac{\beta_{\bar{y}_1}(\bar{y}_1)}{\bar{y}} \models \psi.$$

As φ is finite we can write $\bar{y}_1 = [x_1, \ldots, x_n]$ and $\bar{y}_1 = [y_1, \ldots, y_m]$.
Let $y \in \bar{y}_1$. Then it holds that $s_1(y) = t$ for some $t \in T$. By the definition of I_s it holds that

$$I_s \frac{\beta_{\bar{x}_1}(\bar{x})}{\bar{x}_1} s_1(\bar{y}_1) = \beta_{\bar{y}_1}(\bar{y}_1).$$

for all $\beta_{\bar{x}_1}$. Therefore

$$I_s \frac{\beta_{\bar{x}_1}(\bar{x}_1)}{\bar{x}_1} \frac{I_s \frac{\beta_{\bar{x}_1}(\bar{x}_1)}{\bar{x}_1} s(\bar{y}_1)}{\bar{y}_1} \models \psi.$$

Then it follows from the substitution lemma that

$$I_s \frac{\beta_{\bar{x}_1}(\bar{x}_1)}{\bar{x}_1} \models \psi\{\bar{y}_1 / s(\bar{y}_1)\}.$$

and therefore $I_s \models s(\varphi)$. As $[s_2, \ldots, s_k]$ is the skolemisation of $[s_1](\varphi)$ it follows from the inductive hypothesis that I_s is a model of $[s_2, \ldots, s_n]([s_1](\varphi)) = s(\varphi)$.

(d) The proof of (d) is symmetric to that of (a). Again, the direction from left to right follows directly from (b). Let I be a model of $u(\varphi)$ and let $s' = u^{-1}$. Then by (c) there is an interpretation $I_{s'}$ which is a model of $s'(u(\varphi))$. By Lemma 5 it holds that $s'(u(\varphi)) \models\!\!\models \varphi$. Therefore $I_{s'}$ is a model of φ.

□

The following corollary follows directly from the proof of Proposition 8.

Corollary 6.5. Let φ be a finite formula over disjoint sets $A \cup U$. Let u be an un-Skolemisation over U which is applicable to φ. Then it holds that if I is an interpretation over A which is a model of $u(\varphi)$ then there is an extention I_u of I on U which is a model of φ.

6.2 Operational Semantics

6.2.1 A Motivating Example

Reconsider the program P from Example 6.1 in the last section.

$$\exists y_1 \forall x_1 x_2 \exists y_2 .\ p(a, y_1)$$
$$\wedge\ p(x_1, x_2) \to q(x_2, y_2)$$

An induced Skolemisation of this program is $s = [\{y_1 \mapsto c\}, \{y_2 \mapsto f(x_1, x_2)\}]$. Observe that the elements of the minimal Herbrand model of $s(P)$

$$\{p(a, c), q(c, f(a, c))\} \qquad (\star)$$

are entailed by $s(P)$ but not by P itself. To obtain formulas which is entailed by P we need to replace the Skolem terms in these formulas by existential variables. As for example the Skolem terms c occurs in both formulas, we first conjunct the formulas in the above set, to obtain a single formula. Then we can then apply an un-Skolemisation to this formula. Observe that the un-Skolemisation

$$s^{-1} = [\{f(x_1, x_2) \mapsto y_2\}, \{c \mapsto y_1\}]$$

of s is not suitable for this as variables in Skolem terms have been bound to terms during the evaluation of $s(P)$.

Observe that as an *RDFLog* program is range restricted it holds that the semantics of the Skolemisation of an *RDFLog* program is ground. We therefore need to compute the ground instance of s^{-1}:

$$\begin{aligned} gnd(s^{-1}) = [\{&c \mapsto z_1, f(a, a) \mapsto z_2, f(a, c) \mapsto z_3 \\ &f(c, a) \mapsto z_4, f(c, c) \mapsto z_5, f(f(a), a) \mapsto z_6, \ldots\}] \end{aligned}$$

If this Skolemisation is applied to the formula obtained from the conjunction of the formulas in $(\star)$ we obtain

$$\exists z_1 z_3 . p(a, c) \wedge q(c, f(a, c))$$

Observe that this formula is entailed by P. We now formally define the ground instance of an un-Skolemisation.

Definition 35 (Ground Instance of an un-Skolemisation)**.** First consider the following grounding operator *gnd* on terms. Let t be a term. Then

$$gnd(t) = \{t' \mid t' \text{ is a ground instance of } t\}.$$

We extend *gnd* to sets in the usual way. Let $u = [u_1, \ldots, u_n]$ be an un-Skolemi-sation. Let $\{t_1, \ldots, t_m\}$ be the set of ground instances of terms in the domain of u, i.e.

$$\{t_1, \ldots, t_m\} = gnd(dom(u_1)) \cup \ldots \cup dom(dom(u_n))$$

Finally let $z_1, \ldots, z_m$ be fresh variables. Then the *ground instance* of u is the un-Skolemisation

$$[\{t_1 \mapsto z_1, \ldots, t_m \mapsto z_m\}]$$

6.2.2 Definition of the Operational Semantics

We can now define the operational semantics of *RDFLog*.

Definition 36 (Operational Semantics of *RDFLog*)**.** Let P be an *RDFLog* program. Let s be a Skolemisation over S which is induced by P and such that S is disjoint with the *RDF* alphabet. Then *operational semantics* of P is defined as

$$[\![P]\!]^o = \{u([\![s(P)]\!]_I) \mid s \text{ is a Skolemisation of } P \text{ and } u = gnd(s^{-1})\}.$$

6.2.3 Why RDFLog Programs need to be Range Restricted

Having defined the operational semantics of *RDFLog*, it is now possible to accept

- why *RDFLog* programs need to be range restricted and
- why our definition of the Skolemisation does not replace existential variables by terms with distinct function symbols.

The second point is easy to see. We want the inverse of un-Skolemisations to be Skolemisations. If we do not use our more general definition, the inverse of ground instances of un-Skolemisations are no Skolemisations.

The first point is connected to the second. If an *RDFLog* program P is not range restricted, then the semantics of $s(P)$ is not ground. Consider the following example of a non-range restricted *RDFLog* program.

$$\begin{aligned}\forall x_1 x_2 \exists y_1 \forall x_3.\ & p(a, x_3) \\ & \wedge\ p(a, b) \\ & \wedge\ p(x_1, x_2) \rightarrow q(x_1, y_1)\end{aligned}$$

The semantics of $s(P)$ is

$$\forall x.\ p(a, x) \wedge q(a, f(a, x)) \wedge q(a, f(a, b))$$

To un-Skolemise this formula we need replace both $f(a, x)$ and $f(a, b)$ by existential variables. But as these two terms unify this operation is not an un-Skolemisation by our definition. As discussed in the previous section, an operation which maps unifying terms to existential variables does not have the properties of our un-Skolemisation. We need these properties in order to make our definition of the denotational semantics hold. This semantics will be defined in the next chapter.

Chapter 7

Denotational Semantics

In this section we define the denotational semantics of RDFLog. We first discuss some properties of the operational semantics $[\![P]\!]^o$ of an RDFLog program P in Section 7.1). These properties are useful to define the denotational semantics $[\![P]\!]^d$ of P in Section 7.2. This semantics characterises the operational semantics up to logical equivalence. In Section 7.3 we present a semantics of P which is (i) minimal and (ii) unique up to renaming of variables. Nonetheless we show these properties have a computational cost.

7.1 Properties of the Operational Semantics

In this section we show tree properties of the operational semantics $[\![P]\!]^o$ of an RDFLog program P.

(a) The operational semantics $[\![P]\!]^o$ is entailed by P.

(b) Every RDF graph, which is entailed by P is also entailed by $[\![P]\!]^o$.

(c) The interpretation related to $[\![P]\!]^o$ is a model of P.

As a first approximation to a denotational semantics of RDFLog consider the set of all RDF graphs which are entailed by a RDFLog program P.

$$\{g \in RDF \mid P \models g\}$$

One could say that the property (a) is the soundness of RDFLog w.r.t. ground entailment and properties (b) and (c) are both completeness properties. Nonetheless we can give a more precise characterisation of the operational semantics. The ground entailment is not even unique up to logical equivalence: the empty RDF graph (which corresponds th the logical formula *true*) is in the ground entailment of every RDFLog program. These properties are nonetheless useful: We show in the next section that the conjunction of (a) and (b) and the conjunction of (a) and (c) characterise $[\![P]\!]^o$ up to logical equivalence.

7.1.1 Operational Semantics is not to Big

To show that the operational semantics of RDFLog has property (a) we need the following lemma.

Lemma 6 (Entailment survives un-Skolemisation). Let φ and ψ be formulas over alphabet $A \cup U$ with A and U disjoint where φ is finite and ψ is ground. Let u be an un-Skolemisation on U which is applicable to φ. Let u' be the ground instance of u. Then it holds for any *RDF* graph $g \in RDF$ that

$$\varphi \models \psi \quad \Longrightarrow \quad u(\varphi) \models u'(\psi)$$

Proof. Let I be a model of $u(\varphi)$. As φ is finite, it holds by Corollary 6.5 that there is an extention I_u of I on U which is a model of φ. By the assumption it holds that I_u is also a model of ψ. Observe that as u' is a ground instance of an Skolemisation on U, and ψ is ground, it holds that $u'(\psi)$ contains no terms with a function symbol in U. As I_u is an extention of I on U it therefore holds that I_u is a model of $u'(\psi)$ iff I is a model of $u'(\psi)$. Therefore I is a model of $u'(\psi)$. □

We can now show statement (a).

Lemma 7. Let P be an *RDFLog* program and s a Skolemisation on S induced by P. Let u be an ground instance of s^{-1}. Then it holds that

$$P \models u([\![s(P)]\!]_l)$$

Proof. As $s(P)$ is a logic program, it follows from Lemma 4.2 that $s(P) \models [\![s(P)]\!]_l$. As $s(P)$ is finite it follows from Lemma 6 that $u(s(P)) \models u([\![s(P)]\!]_l)$. Finally by Lemma 5 it holds that $P \models\!\!\models u(s(P))$ and therefore $P \models u([\![s(P)]\!]_l)$. □

7.1.2 Operational Semantics is not to Small

To show statement (b) we need the following Lemma.

Lemma 8. Let P be a logic program and g an *RDF* graph. Then it holds that

$$P \models g \implies [\![P]\!]_l \models g$$

Proof. As $[\![P]\!]_l$ is defined to be the conjunction of all atomic formulas which are entailed by P it is clear that the statement if true if g contains no existential variables.

Let $\bar{x}$ be the blank nodes occurring in g. Observe that by proposition 4 it holds that there is a sequence of terms $\bar{t} \in U_P$ such that $P \models g\{\bar{x}/\bar{t}\}$. As $g\{\bar{x}/\bar{t}\}$ is entailed by P, it is a subformula of $[\![P]\!]_l$. Therefore $[\![P]\!]_l \models g\{\bar{x}/\bar{t}\}$. The proposition follows from the fact that $g\{\bar{x}/\bar{t}\} \models g$. □

We can now show (b).

Lemma 9. Let P be an *RDFLog* program. Let s be a Skolemisation over S which is induced by P and such that S is disjoint with the *RDF* alphabet. Let u be an ground instance of s^{-1}. Then it holds that

$$\forall h \in RDF. P \models h \implies u([\![s(P)]\!]_l) \models h$$

Proof. Assume that $P \models h$. As P is finite, it follows from Lemma 8 that $s(P) \models h$ and from Lemma 8 that $[\![s(P)]\!]_l \models h$. As P is range restricted it holds that $[\![P]\!]_l$ is ground. Therefore we can use Lemma 6 to conclude that $u([\![s(P)]\!]_l) \models u(h)$. As $h \in RDF$ and S is disjoint with the *RDF* alphabet it holds that $u(h) = h$. Therefore $u([\![s(P)]\!]_l) \models h$. □

7.1.3 Operational Semantics is a Model

Recall what we mean by an interpretation *related* to an RDF graph (Section 3.3). Using these notion we can show statement (c).

Proposition 9. Let P be an RDFLog program and let $I = I_{[\![P]\!]^o}$ be the interpretation related to $[\![P]\!]^o$. Then I is a model of P.

Proof. Let s be a skolemisation of P. As $s(P)$ is a logic program it holds that $M_{s(P)}$ is a model of $s(P)$. As $s(P)$ entails P (Proposition 8) it holds that $M_{s(P)}$ is also a model of P. As no symbols from S occur in P it holds that the restriction of $M_{s(P)}$ to the symbols of A is a model of P. It is easy to see that this interpretation is isomorphic to the interpretation related to $I_{[\![P]\!]^o}$. □

7.2 Charachterisation up to Equivalence

In this section we show that the three properties of an RDFLog program P

(a) the operational semantics $[\![P]\!]^o$ is entailed by P,

(b) every RDF graph, which is entailed by P is also entailed by $[\![P]\!]^o$, and

(c) the interpretation associated with $[\![P]\!]^o$ is a model of P.

allow to characterise the operational semantics of RDFLog up to equivalence. To be precise we show that the set M of RDF graphs which have both properties (a) and (b) is an equivalence class. We give an example that shows that the set N of RDF graphs with properties (a) and (c) are *no* equivalence class. Nonetheless if g and h are RDF graphs with properties (a) and (b) then they are equivalent. This shows that the set N is a subset of M. We define the denotational semantics of an RDFLog program to be N. We show how the soundness and completeness of the operational semantics w.r.t this denotational semantics follow from the results of the last Section. Properties (a) and (b) can be seen as a second characterisation of the operational semantics up to equivalence.

7.2.1 Possible Characterisations

Lemma 10. The set M

$$\{g \in RDF \mid P \models g \wedge \forall h \in RDF \,.\, P \models g \Rightarrow g \models h\}$$

is an equivalence class.

Proof. Let $g, h \in M$. Therefore $P \models g$. From this it follows that $h \models g$. Now let $g \in M$ and $h \models g$. It follows directly that $h \in M$. □

Using Lemma 4 it is easy to show that the RDF graphs with properties (a) and (c) are equivalent.

Lemma 11. Let g, h be RDF graphs in the set N

$$\{g \in RDF \mid P \models g \wedge I_g \vDash P\}.$$

Then $g \models h$.

Proof. Let $g, h \in N$. As $I_g \models P$ and $P \models h$ that $I_g \models h$. It follows from Lemma 4 that $g \implies h$. Now let $g \in N$ and $h \models\!\!\!| g$. □

Observe thought that the set N is *not* an equivalence class. Consider the following example

Example 7.1. Let P be the RDFLog program

```
p(a,b)
∀ x . p(a,x) → q(a,x)
```

Let g and h be RDF graphs where $g = p(a,b) \wedge q(a,b)$ and $h = \exists x \,.\, p(a,x) \wedge p(a,b) \wedge q(a,b)$. It is easy to see that $g \models\!\!\!| h$ and that $I_g \models P$. Observe that $I_h = (D, i, \varnothing)$ where $D = \{a, b, d\}$ and

$$a^i = a \qquad p^i = \{(a,b),(a,d)\}$$
$$b^i = b \qquad q^i = \{(a,b)\}$$

It is easy to see that $I_h \not\models P$.

The previous example shows that the set N is a subset of the set M from Lemma 10. Nonetheless this is still not a characterisation up to isomorphism: The RDF graph $\exists x \,.\, p(a,x) \wedge q(a,x) \wedge p(a,b) \wedge q(a,b)$ is not isomorphic to g but is in the set N for the program from example 7.1.

7.2.2 Definition of the Denotational Semantics

These results show that both sets, N and M, could be used as semantics for RDFLog. We choose N as it is the smaller set.

Definition 37 (Denotational Semantics of RDFLog)**.** Let P be an RDFLog program. The *denotational semantics* of P is defined as

$$[\![P]\!]^d := \{g \in RDF \mid P \models g \wedge I_g \models P\}$$

where I_g is the interpretation related to g.

7.2.3 Soundness and Completeness of the Denotational Semantics

The soundness and completeness of RDFLog are immediate from the results we have obtained in the previous sections.

Proposition 10 (Soundness and Completeness of RDFLog). Let P be an RDFLog program. The operational semantics $[\![P]\!]^d$ is sound and complete wrt. the operational semantics $[\![P]\!]^d$.

Proof.

- *Soundness* We need to show that $[\![P]\!]^o$ is an element of $[\![P]\!]^d$. This is immediate from Lemma 7 and Lemma 9.
- *Completeness* We need to show that any element of $[\![P]\!]^d$ is entailed by $[\![P]\!]^o$. It follows from the fact that $[\![P]\!]^d$ contains only equivalent RDF graphs (Lemma 11) and that $[\![P]\!]^o$ is an element of $[\![P]\!]^o$ (Soundness of RDFLog) that $[\![P]\!]^o$ is equivalent to any element of $[\![P]\!]^d$.

□

Now consider the set M

$$\{g \in RDF \mid P \models g \wedge \forall h \in RDF \,.\, P \models g \Rightarrow g \models h\}.$$

We showed that $[\![P]\!]^o$ is an element of M in Lemma 9 and Lemma 9. In this section we showed that M is an equivalence class (Lemma 10). Therefore M can be seen as a second characterisation of the operational semantics of RDFLog up to equivalence.

Proposition 11. Let P be an RDFLog program. Then it holds that

$$[\![P]\!]^d \subseteq \{g \in RDF \mid P \models g \wedge \forall h \in RDF \,.\, P \models g \Rightarrow g \models h\}.$$

7.3 Charachterisation up to Isomorphism

The definition of the denotational semantics in the last Section is somewhat unsatisfactory: The operational semantics is only characterised up to logical equivalence. Still any model of an RDFLog program P is an element of the denotational semantics if its related RDF graph is entailed by P. It is desirable to have a characterisation of the operational semantics up to isomorphism.

7.3.1 Minimal Semantics for RDFLog

Recall that there is a similar dilemma in logic programming: a logic program has an infinite number of models in the general case. The dilemma is solved by choosing the minimal of these models. In this section we discuss whether such an approach is suitable for a language like RDFLog.

Reconsider the properties of the operational semantics $[\![P]\!]^o$ of an RDFLog program P mentioned in the previous sections.

(a) P entails $[\![P]\!]^o$,

(b) all RDF graphs which are entailed by P are also entailed by $[\![P]\!]^o$, and

(c) the interpretation related to $[\![P]\!]^o$ is a model of P.

In this section, we check whether the set consisting of all minimal RDF graphs with one of these properties also has the other properties. If this is not the case we consider this set as not suitable as a definition of the semantics.

Lets start with property (a). It is easy to see that the empty RDF graph is entailed by any RDFLog program P. On the other hand it is obvious that the empty RDF graph has neither property (b) nor property (c) and is therefore not suited as a semantics of P.

Now consider property (c) as it is the one used in the definition of the semantics of logic programming. The following example shows why the set M

$$\min\{g \in RDF \mid I_g \text{ is a model of } P\}$$

is not suitable as a semantics for RDFLog:

Example 7.2. Consider the RDFLog program P

```
∃ x . p(a,x) ∧ p(b,x).
p(a,c).
```

Observe that this is an example of an RDFLog program which is an RDF graph. A minimal model for P is $I = (D, i, \beta)$ where $D = \{a, b, c\}$ and

$$a^i = a \qquad b^i = b$$
$$c^i = c \qquad p^i = \{(a, c), (b, c)\}$$

The RDF graph related to I is $g = p(a, c) \wedge p(b, c)$. It is easy to see that g is not entailed by P. Nonetheless g has property (b). The following example shows that this is not always the case:

Example 7.3.

```
∃x₁x₂ . p(a,x₁) ∧ q(a,x₂).
```

In this case a minimal model for P is $I = (D, i, \beta)$ where $D = \{a, d\}$ and

$$a^i = a \qquad p^i = q^i = \{(a, d)\}$$

Observe that the RDF graph g related to I is $\exists x \,.\, p(a, x) \wedge q(a, x)$. Therefore property (b) does not hold: As P is an RDF graph and it is entailed by itself it should be entailed by g. This is clearly not the case. Even though this example does have property (a) it is easy to construct an RDFLog program P such that the set of minimal models of P has neither property (a) nor (b).

Lets finally turn to property (c). It is actually not known to the author whether the set N of minimal RDF graphs which entail all RDF graphs g which is entailed by an RDFLog program P is has properties (a) and (c). We conjecture that this is true but leave it as an open question.

To sum up recall that we showed that the properties (a) and (b) are not strong enough to define a sensible minimal semantics for RDFLog. It is not known whether this is the case for property (c). Nonetheless it is known from the last section that the set of RDF graphs with properties (a) and (b) is an equivalence class. In addition it is known from the literature [35] that an equivalence class of RDF graphs have a unique up to isomorphism representative: the lean graph. This observation motivates our definition of a semantics for an RDFLog program P which is minimal and unique up to isomorphism:

Definition 38 (Denotational Lean Semantics of RDFLog)**.** Let P be an RDFLog program. Then the *denotational lean semantics* $[\![P]\!]^d_{lean}$ of P is defined as

$$[\![P]\!]^d_{lean} = \{g \in RDF \mid g \text{ has properties (a) and (b), and } g \text{ is lean}\}$$

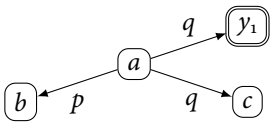
7.3.2 Naive Operational Lean Semantics

It is easy to see that the operational semantics of RDFLog is not an element of the lean semantics (simply consider an RDFLog program consisting of a non-lean graph and no rules). Even worse an RDFLog program may turn an lean graph into an non-lean one. Consider the following example:

Example 7.4.

```
p(a,b).
q(a,c).
∀ x ∃ y . p(a,x) → q(a,y)
```

Observe that the operational semantics of this program is the following RDF graph



Therefore we develop an operational semantics for the lean semantics. Recall that an RDF graph g is lean iff there is no subgraph h of g which entails g. By the interpolation lemma this is equivalent to the fact that there is no substitution σ for the existential variables in g such that $g\sigma$ is a subgraph of h. This observation motivates the definition of the following operator

Definition 39 (Lean Operator). Let g be an RDF graph and Σ a set of substitutions. The *lean operator* is defined as

$$leaner_\sigma(g) := \begin{cases} g\sigma & \text{if } g\sigma \subseteq g \\ g & \text{otherwise} \end{cases}$$

$$L_\Sigma(g) := \min\{leaner_\sigma(g) \mid \sigma \in \Sigma\}$$

It follows from Proposition 1 that an RDF graph g is lean if $g \in L_\Sigma(g)$ and Σ is the set of all substitutions from blank nodes in g to nodes in g. As the application L preserves logic equivalence it is easy to see that its application also preserves properties (a) and (b). Therefore the naive lean semantics of RDFLog defined below is sound and complete

Definition 40 (Naive Lean Semantics of RDFLog). Let P be an RDFLog program and let Σ be the set of all substitutions from blank nodes in $[\![P]\!]^o$ to nodes in $[\![P]\!]^o$. The *naive operational lean Semantics* $[\![P]\!]^o_{lean}$ of P is defined as

$$[\![P]\!]^o_{lean} := L_\Sigma\left([\![P]\!]^o\right)$$

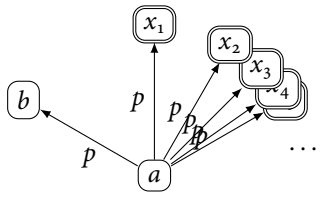
7.3.3 Less Naive Operational Lean Semantics

There are cases in which the operational semantics of RDFLog is infinite while the lean semantics is finite. Consider for example the following program.

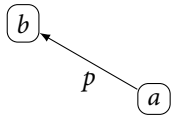
Example 7.5.

```
p(a,b).
∀ x ∃ y . p(a,x) → p(a,y)
```

Observe that the operational semantics of this program is infinite. A subset is depicted below.



The lean version of this graph has only one edge.



Therefore it is desirable to make intermediate results lean. In the previous example the following algorithm might be conceivable. After every application of the consequence operator the resulting intermediate result is made lean. Consider a sequence of RDF graphs $g_0, g_1, \ldots$ such that g_n is the result of n-fold application of a consequence and lean operator (which we define below). In the previous example in the first application of the consequence operator yields the set $g_0 = \{p(a, b)\}$. Next, the operator L is applied to g_0. As g_0 is lean this has no effect. In the next step the fact $\exists x \,.\, p(a, x)$ is added, yielding $g_1 = \{\exists x \,.\, p(a, b) \wedge p(a, x)\}$. This time g_1 is not lean and $L(g_1)$ is $\{p(a, b)\}$. As this set is equal to g_0 the algorithm terminates.

We now give a precise definition of the algorithm. Recall that the consequence operator for logic programs.

$$cons_r(M) := \begin{cases} M \cup \{a\} & \text{if } r = a_1 \ldots a_n \to a \text{ and } \{a_1 \ldots a_n\} \subseteq M \\ M & \text{if } r = a_1 \ldots a_n \to a \text{ and } \{a_1 \ldots a_n\} \nsubseteq M \end{cases}$$

$$T_P(M) := \bigcup_{r \in gnd(P)} cons_r(M)$$

where P is a logic program and M is a set of atoms.

We use this definition of *cons* to define a lean consequence operator. We show that this operator does not preserves completeness in the general case. Then we present a condition which ensures that completeness is preserved.

Definition 41 (Lean Consequence Operator)**.** Let P be a Skolemised RDFLog program, Σ be a set of substitutions and M a set of ground atoms. Then the *lean consequence operator* $LT_{P,\Sigma}$ is defined as

$$LT_{\Sigma,P}(M) := L_\Sigma \circ T_P(M)$$

We often drop the subscript if it is clear or unimportant. The iteration if LT is defined for ordinal numbers as usual.

$$\begin{aligned} LT^0 &= \varnothing \\ LT^\alpha &= LT(LT^{\alpha-1}) && \text{if } n \text{ is a successor ordinal} \\ LT^\alpha &= \bigcup \{LT^\beta \mid \beta < \alpha\} && \text{if } n \text{ is a limit ordinal} \end{aligned}$$

As the foregoing example showed this operator needs to be iterated until a periodic fixed point is reached, i.e. the smallest LT^n with $LT^n = LT^{n-k}$ for some $k > 0$.

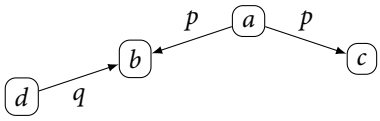
The following example show that the iteration of LT does not preserve completeness.

Example 7.6.

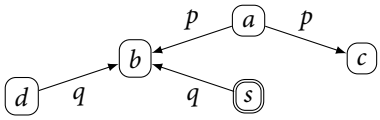
P

```
p(a,b).
p(a,c).
q(d,b).
∃ X ∀ Y . p(a,Y) → q(X,Y).
```

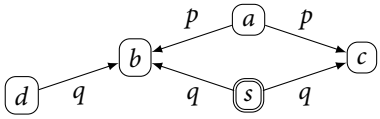
With this program P it holds that $T_P(\varnothing)$ is



As this graph is lean it holds that $L_\Sigma \circ T_P(\varnothing) = T_P(\varnothing)$. After one further application of a T_P the following RDF graph is constructed:



where s is the Skolem symbol that replaces X in the rule in line 4 of P. Observe that if the substitution $\sigma(s) = d$ is applied to the above graph then the result is g_0. Therefore the above sketched algorithm terminates. Nonetheless the operational semantics of the above program is the following graph g which is not lean.



As the interpretation related to g_0 is not a model of P the above sketched algorithm is incomplete w.r.t. our definition of the denotational semantics.

We now give two sufficient conditions under which intermediary results may be made lean without affecting completeness. We first give a Lemma without a proof. The intuition is that if no edges are added to an blank node in an RDF graph g then an g will not become lean in the sequel.

Lemma 12. Let g be an RDF graph, r an RDFLog rule, and σ be a substitution from blank nodes to terms in g. If no universal quantifier appears in the scope of an existential quantifier in r then $leaner_\sigma$ is commutative with $cons_r$, i.e.

$$leaner_\sigma \circ cons_r(g) = cons_r \circ leaner_\sigma(g).$$

Observe that the program in Example 7.6 does not satisfy this condition. We now work towards a second sufficient condition for commutativity of *leaner* and *cons*.

Lemma 13. Let $g = \bigwedge G$ be an skolemised RDF graph and let σ be a substitution. Let $r = a_1, \ldots, a_n \rightarrow a$ be a ground Skolemised RDFLog rule and $A = \{a_1, \ldots, a_n\}$. If the terms in r and the domain of σ are disjoint then it holds that

$$leaner_\sigma \circ cons_r(g) = cons_r \circ leaner_\sigma(g)$$

Proof.

We first show the statements (a) - (d) which we need in the rest of the proof.

(a) $A \subseteq G\sigma \Rightarrow A \subseteq G$: Let $t \in A$. Then it follows from the hypothesis that $t \in G\sigma$. As $t \in A \subseteq T$ it holds that $t\sigma = t$. Therefore it holds that $t \in G$.

(b) $G\sigma \subseteq G \Rightarrow (G \cup \{a\})\sigma \subseteq G \cup \{a\}$: Let $t \in (G \cup \{a\})\sigma$. If $t \in G\sigma$ then $t \in G$ by hypothesis. Now assume that $t = a\sigma$. As $dom(\sigma) \cap T = \varnothing$ it holds that $a\sigma = a$. Therefore $t = a$.

(c) $a\sigma = a$: Follows directly from the hypothesis.

(d) $A \subseteq G \Rightarrow A \subseteq G\sigma$: Let $t \in A$. By the hypothesis it follows that $t \in G$. As the terms in r and the domain of σ are disjoint it holds that $t \in G\sigma$.

We can now show the lemma.

- If $A \nsubseteq G$ and $G\sigma \nsubseteq G$ then it holds that $leaner_\sigma \circ cons_r(g) = g = cons_r \circ leaner_\sigma(g)$.
- If $A \nsubseteq G$ and $G\sigma \subseteq G$ then $leaner_\sigma \circ cons_r(g) = leaner_\sigma(g) = g\sigma \overset{(a)}{=} cons_r(g\sigma) = cons_r \circ leaner_\sigma(g)$.
- If $A \subseteq G$ and $G\sigma \nsubseteq G$ and
  - if $(G \cup \{a\})\sigma \nsubseteq G \cup \{a\}$ then it holds that $leaner_\sigma \circ cons_r(g) = leaner_\sigma(g \cup a) = g \cup a = cons_r(g) = cons_r \circ leaner_\sigma(g)$ and otherwise
  - if $(G \cup \{a\})\sigma \subseteq G \cup \{a\}$ then it holds that $leaner_\sigma \circ cons_r(g) = leaner_\sigma(g \cup a) = (g \cup a)\sigma \overset{\text{Proposition 1}}{\models} g \cup a = cons_r(g) = cons_r \circ leaner_\sigma(g)$.
- If $A \subseteq G$ and $G\sigma \subseteq G$ then it holds that $leaner_\sigma \circ cons_r(g) = leaner_\sigma(g \cup a) \overset{(b)}{=} (g \cup a)\sigma = g\sigma \cup a\sigma \overset{(c)}{=} g\sigma \cup a \overset{(d)}{=} cons_r(g\sigma) = cons_r \circ leaner_\sigma(g)$.

□

During a forward evaluation of a rule based program it is possible to distinguish rules that will not be needed for the evaluation anymore. A simple example would be a rule with no variables in the head which have already been used. It is possible to develop more elaborate conditions, but we do not address this topic here. We assume that given a RDFLog program P there is a set of *active* rules which may still be needed for evaluation.

Proposition 12. Let P be an RDFLog program and let s be a Skolemisation of P. Let Σ be the set of all substitutions σ such that the domain of σ contains no terms which are instances of Skolem terms in active rules. Then it holds that the un-Skolemisation of any periodic fixed point of $LT_{s(P),\Sigma}$ is logically equivalent to the naive lean semantics of RDFLog.

Proof. We need to show that if Σ is as above then a the un-Skolemisation of a periodic fixed point of $LT_{s(P),\Sigma}$ is equal to $L_\Gamma\left(\llbracket P \rrbracket^o\right)$ where Γ is the set of all substitutions from blank nodes in $\llbracket P \rrbracket^o$ to nodes in $\llbracket P \rrbracket^o$. Recall that $\llbracket P \rrbracket^o$ is $\{u(\llbracket s(P) \rrbracket_l) \mid s \text{ is a Skolemisation of } P \text{ and } u \text{ is a grounding of } s^{-1}\}$ and that $\llbracket P \rrbracket_l$ is the conjuction of the atoms in least fixed point of T_P.

Let g be an RDF graph and s a Skolemisation of g and $u = s^{-1}$. First observe that $L_\Gamma \circ u(g) = u \circ L_\Sigma(g)$ if $\Gamma = \{\sigma u \mid \sigma \in \Sigma\}$. Therefore it is sufficient to show that a periodic fixed point of $LT_{s(P),\Sigma}$ is logically equivalent to $L_\Gamma(T^n_{s(P)})$ where $T^n_{s(P)}$ is the least fixed point of $T_{s(P)}$.

We show this statement by induction on n. We denote $T_{s(P)}$ by T, L_Γ by L and $LT_{s(P),\Sigma}$ by LT.

- $n = 0$: We need to show that a periodic fixed point of LT is logically equivalent to $L(T^\omega) = L(\varnothing) = \varnothing$. This is true because as T^ω is a fixed point it holds that $T(\varnothing) = \varnothing$. In addition it holds that $L(\varnothing) = \varnothing$. Therefore (i) $LT^\omega = \varnothing = L(T^\omega)$ and (ii) LT^ω is a periodic fixed point with period 1.

- $n \to n + 1$: By definition it holds that $L(T^{n+1}) = L \circ T \circ T \circ \ldots \circ T(\varnothing)$. As $L \circ L = L$ this is equivalent to $L \circ L \circ T \circ T \circ \ldots \circ T(\varnothing)$. It follows directly from Lemma 13 that this is equal to $L \circ T \circ L \circ T \circ \ldots \circ T(\varnothing)$ and this in $LT \circ L \circ T \circ \ldots \circ T(\varnothing)$. By the hypothesis it follows that this is equivalent to LT^n. The same argument as in the case where $n = 0$ shows that LT^n is a periodic fixed point with period 1.

□

Chapter 8

Expressivity and Complexity

In this section, we extend the gained insight beyond RDFLog and summarize the gains (in expressiveness) and the costs (in complexity) when adding b-nodes to an RDF query language. We introduce a hierarchy of options for increasingly more expressive b-node support. For each of this options we study expressiveness and complexity under combinations of lean semantics and recursion.

8.1 Lean semantics

Recall, the definition of lean semantics from Definition 15 in Section 3.1. For our purpose, the most important result is due to [30]:

Theorem 1 (Complexity of lean answer semantics). Given an *RDFLog* query q and an RDF graph D, computing the answer for q on D under lean answer semantics is DP-hard[1] (i.e., both NP-hard and CO-NP-hard).

Proof. Follows by reduction from computing the core of a relational structure which is shown to be DP-hard in [30]. CO-NP-hardness is also shown independently in [36]. □

Unfortunately, the complexity stays the same if D and q are lean (where we understand q to be lean if there is an input graph D' such that the answer for q on D' is lean under standard answer semantics).

8.2 Recursion

Limiting recursion in a rule-based language such as *RDFLog* is a natural choice for a less expressive sublanguage with better computational properties. The essential observation when considering the effect of recursion on *RDFLog* is that b-nodes can be used to simulate arbitrary terms if combined with recursive rules.

Lemma 14. Let t be a ground term of standard first-order logic. Then there is a set of *RDFLog* rules that constructs a list (in RDF terms: collection) representation of t.

[1] Recall, that DP is the class of all decision problems, that can be expressed as the intersection of an NP- and a CO-NP-problem.

recursion	no		yes	
lean semantics	no	yes	no	yes
none	P	—	P-C.	—
	PSPACE-C.	—	EXPTIME-C.	—
only in facts	P	DP-C.	P-C.	DP-C.
	PSPACE-C.	PSPACE-C.	EXPTIME-C.	EXPTIME-C.
only independent	P	DP-C.	P-C.	DP-C.
	PSPACE-C.	PSPACE-C.	EXPTIME-C.	EXPTIME-C.
single-rule, dependent	P	DP-C.	—	—
	PSPACE-C.	PSPACE-C.	—	—
multi-rule, dependent	P	DP-C.	*r.e.*-C.	*r.e.*-C.
	NEXPTIME-C.	NEXPTIME-C.	*r.e.*-C.	*r.e.*-C.

Table 1. Combining b-nodes and recursion: Complexity for each class of b-node support the first line gives data complexity, the second query complexity. The 'c.' is a shorthand for 'complete'

Sketch. If t is a constant c, we use $(id_t \mathsf{represents} c)$ as head of a single rule. If t is composite, let $t = f(t_1, \ldots, t_n)$. The list representation of t is $[f, id_{t_1}, \ldots, id_{t_n}]$ where id_{t_1} is the b-node identifying t_1's list representation. This list can be represented, e.g., as RDF collection. □

Note, that for the construction of collections we need b-nodes in the scope of some universal variables. Independent b-nodes, i.e., b-nodes outside the scope of any universal variable, do *not* suffice.

From this lemma it follows directly that we can simulate a two-stack machine [46] in *RDFLog*. Thus, like full logic programming, *RDFLog* is *r.e.*-complete.

Theorem 2 (Recursion and b-nodes). Let L be a rule-based RDF query language with recursion and b-nodes such as *RDFLog*. Then L is *r.e.*-complete.

8.3 Classification of b-node support

Table 1 gives an overview of the different classes of b-node support and their complexity in different combinations with lean semantics and recursion. In the following, we give brief summaries of the differences between the classes.

8.3.1 Independent b-nodes

The most basic form of support for b-nodes is to allow them only to occur *independent* of any universal variable. We call a b-node dependent of another variable, if it is in the scope of that variable. Otherwise we call it independent. Finally, we call a b-node fully-dependent in a rule r if it depends on all universal variables of r.

B-nodes *only in facts*, i.e., in RDF graphs, is a special case of this limitation, as facts are range-restricted and thus contain no universal variables.

In either case, this limitation is indeed computationally attractive: It guarantees that the number of b-nodes is limited by the number of rules (and facts). Therefore, it has no effect on either expressiveness or complexity (which remain the same as relational algebra, resp. Datalog, if recursion is present) unless we are also interested in lean answer semantics. Under lean answer semantics, the data complexity increases to DP-complete, see Theorem 1.

8.3.2 Non-recursive single-rules

At first surprising is the effect of limiting a language to single-rule programs. If we allow disjunction in rule bodies, non-recursive, range restricted single-rule programs are equivalent to non-recursive, range restricted multi-rule programs and thus to relational calculus if no b-nodes are allowed.

If dependent b-nodes are allowed, however, single-rule programs with disjunction are no longer equivalent to multi-rule programs even if non-recursive and range restricted. The reason is that multi-rule programs allow repeated introduction of new dependent b-nodes. In particular, b-nodes may depend on b-nodes introduced by another rule.

Theorem 3. Non-recursive, range-restricted multi-rule programs are NEXPTIME-complete wrt. query complexity, whereas non-recursive, range-restricted single-rule programs are PSPACE-complete wrt. query complexity.

Sketch. The former follows by reduction from non-recursive logic programming. The latter follows by reduction from relational calculus with value invention which has the same complexity as relational calculus without value invention [48]. □

SPARQL is, in fact, an example of a language restricted to single-rule programs but with dependent b-nodes and thus is P-complete wrt. data and PSPACE-complete wrt. program complexity. This reinforces and extends independent complexity results in [63] where PSPACE-completeness is shown for SPARQL patterns, i.e., SPARQL queries *without* graph construction in the head. Our results here show that SPARQL remains PSPACE-complete even when considering full SPARQL queries instead of patterns only.

Another consequence of this result is that SPARQL could be extended from only fully-dependent b-nodes to arbitrary (dependent) b-nodes in the head without sacrificing complexity, but gaining the ability to express additional grouping queries.

8.3.3 Dependent b-nodes

The next stronger class allows arbitrary dependent and independent b-nodes. Interestingly, this class's languages are in effect equivalent to rule-based languages with value invention as introduced, e.g., in [1]. Value invention is an extension of relational query languages for overcoming the expressiveness limitations of Datalog and similar formalisms. Though value invention does not increase the expressive power of relational algebra or calculus [48] it does increase the expressive power if added to a Datalog-like language such as *RDFLog*. Value invention in the sense of [1] corresponds to dependent b-nodes and has essentially the same complexity and expressiveness as logic programming with function symbols (recall Lemma 14).

However, *RDFLog* with dependent b-nodes and recursion differs in one important point from Datalog with value invention: An answer to an *RDFLog* query is a formula involving existential variables rather

than merely a conjunction of ground atoms as in standard logic programming and in [1]. Answers are therefore inherently equivalent only up to an isomorphism on the existential variables.

For logic languages with value invention, the effect of value invention on genericity has been studied extensively [1, 20]. Recall, that a query is called generic if its outcome is independent of the constants in the domain that do not occur in the query.

Since the values invented by value invention operators in [1, 20] are treated like any other constant, genericity is violated if such values occur in the result. This has lead to the definition of restrictions on logic languages with value invention that ensure that invented values are never part of a query answer [20].

An answer to an *RDFLog* query, however, distinguishes between proper constants (URIs in the RDF sense) and existential variables (which may be "invented" during program evaluation). This allows *RDFLog* queries to be generic without disallowing "invented" values, i.e., existential variables in the answer. The price *RDFLog* pays for this is the unskolemisation of answers needed after the standard logic programming evaluation. However, this price is, small, indeed, as unskolemisation is a linear operation in the size of the answer. However, there is a second, less obvious price: Since *RDFLog* answers are formulas involving existential quantifiers equivalence (and thus answer minimization) is no longer a cheap operation but subsumes to finding the core of an *RDFLog* answer, a DP-complete operation. If we are interested in minimal or lean answers, the data complexity of non-recursive *RDFLog* increases accordingly.

Chapter 9

Related Work

In this chapter we investigate the blank node support of several RDF query languages. Further information on this topic can be found in the survey papers [9, 32, 37].

9.1 RQL and SeRQL

RQL [50, 51] is an RDF query language for both RDF and RDF Schema, based on the syntax of OQL [2]. Its most distinguishing feature is its capability to smoothly combining shema and data querying.

RQL is a functional language: a set of basic queries and iterators, which may be used to build new ones through functional composition. Answers are reported as variable bindings. Features supported by RQL are typing, grouping, aggregation, and arithmetic. An interesting feature is the support for generalized path expressions, which allows to navigate through an RDF graph. It should be noted that the semantics of RQL is not completely compatible to the RDF semantics: a number of restrictions are placed on RDF graphs.

The query language SeRQL [17, 18] is an RDF query language designed to gain from the experience from languages such as RQL, RDQL [61, 71, 21] and Notation 3 [12]. The authors of SeRQL propose the following requirements for RDF query language: expressive power, schema awareness, to ease meta programming, compositionality and to have a precis semantics. Strangely the authors do not consider blank node construction to be necessary for an expressive RDF query language.

The result of an RQL query is a set of variable bindings. Therefore there is no construction of blank nodes.

9.2 TRIPLE

TRIPLE [27, 72] is an rule based RDF query language based of Horn logic and F-logic [52]. Languages on top of RDF, such as RDFS and Topic maps can be defined by TRIPLE rules. In addition TRIPLE also provides access to external programs implementing e.g an DAML-OIL reasoner. TRIPLE has native support for namespaces, abbreviation, reification, rules with full first order bodies. Its syntax is inherited

from F-logic. Like RQL and SeRQL it supports path expressions. It should be noted that the mapping from RDF to TRIPLE is not lossless as blank nodes are made explicite.

TRIPLE has a language construct called reification. This construct is more similar to reification in logic programming, than to reification in the context of RDF. An atom is reified by constructing a term representing this atom.

TRIPLE has no support for blank nodes. The semantics of TRIPLE is defined by a translation to Horn logic. The authors refer to a future document where a model theoretic semantics based on minimal Herbrand models will be provided.

9.3 Flora-2

FLORA-2 [81, 76, 77, 78] is a rule based, object oriented RDF query language based of HiLog [24, 23], F-logic, and transaction logic [16, 15]. An Flora-2 program is a set of Flora-2 rules of the form `Head :- Body`, where `Head` is an atom and `body` is any first order formula. The semantics of Flora-2 is defined in [79].

Flora-2 supports constructions of blank nodes, but in a more restricted way than RDFlog. Blank nodes may occur in the head of rules but their (implicit) quantifiers are to the left of universal quantifiers. Therefore Flora-2 corresponds a syntactical restriction of RDFLog.

Yang and Kifer present a fixed point semantics of an abstraction of FLORA-2, called F-logic [80]. The fixed point operator of a F-logic program P is equal to the fixed point operator of logic programming for the skolemisation of P. They state that the least fixed point of their operator is a minimal model of an F-logic program. This result is somewhat surprising if one considers for example the F-logic programs P consisting only of the facts $p(a, b)$ and $\exists x \,.\, p(x, b)$. In this case the minimal Herbrand model of the Skolemisation of P has two atoms. It is easy to see that there is a smaller model of P.

The reason for this lies in the definition of a model in [80]. This deviates from the definition of a model in logic and in RDF in two ways: An interpretation I is a model of a formula φ in the sense of [80] iff I is a model of the Skolemisation of φ in the sense of logics or RDF. The second difference is that the Skolem symbols need to be mapped to domain elements disjoint from the other symbols. Hence for example $p(a, b)$ does not imply $\exists x \,.\, p(x, b)$ with this definition, while this is true under logical and RDF entailment.

9.4 SPARQL

SPARQL [65] is a RDF query language based on SquishQL [61] and RDQL [71]. It is a W3C Candidate Recommendation since the 17 of June 2007. Its syntax is based on Turtle [11].

Following [64] a SPARQL query consists of three parts: a pattern matching part with features like optional matching, union, nesting, filtering, and the possibility to query multiple data sources. Solution modifiers such as projection, distinct, order, limit and offset. Thirdly SPARQL supports four different output types: boolean answers, variable bindings, construction of RDF graphs, and descriptions. The exact semantics of descriptions is not defined in [65], but concise bounded descriptions [74] are referenced as a reasonable choice. Variable bindings may be represented by a special XML document [10].

SPARQL has ample blank node support, but still more restricted than RDFLog, and sub-optimal with respect to expressivity as we will show. In SPARQL blank nodes may occur in queries. These blank nodes

correspond to universal variables in the body of a Datalog rule, which do not occur in the head.

Blank nodes may be constructed implicitly by using the special syntactic short hand notations like '`(`' and '`)`' for RDF collection construction: E.g. `(1, ?x)` is a shorthand notation for

```
 _:b0  rdf:first  1 ;
       rdf:rest   _:b1 .
 _:b1  rdf:first  ?x ;
       rdf:rest   rdf:nil.
_:b0  :p         ''w''
```

In addition, the `describe` keyword allows to extract concise bounded descriptions from RDF graphs.

Blank nodes may also occur in a `construct` part of a SPARQL query. This allows to explicitely construct blank nodes. These blank nodes are 'scoped to the template for each solution' [65]. Therefore a SPARQL program with blank nodes in the construction part corresponds to an RDFLog program who's existential quantifiers are all to the left of the universal quantifiers.

There is also a lot of theoretical work done on the complexity and the Semantics of SPARQL: Perez et. al. show in [64] that the evaluation of SPARQL graph patterns is PSPACE complete even for a SPARQL patterns without filters, RDF/S vocabulary and special treatment for literals. Cyganiak [26] models SPARQL SELECT using relational algebra operators. Using a similar approach Harris [38] presents an implementation of SPARQL queries in a relational database engine. Franconi and Tessaris [31] work on an semantics for SPARQL as an ongoing work.

Part III

Conclusion

Chapter 10

Summary

In this deliverable we investigated how blank nodes can be processed in rule based query languages for RDF. We examine both operational and semantical aspects of such query languages. On the operational side we show how RDF querying can be reduced to logic programming. This makes the results if the research on efficient evaluation of logic programming accessible to RDF querying. On the other hand it is desirable to have a denotational characterisation of the semantics of a programming language. We discuss several possibilities to describe the operational semantics up to logical equivalence. Then we look for a minimal, redundancy free semantics for RDFLog. We show that this semantics can only be computed at certain computational costs. In addition we provide a commutativity result which allows to compute this semantics more efficiently.

In the first sections we introduce RDF from a logical perspective. We recall the definitions from the defining documents [53, 40]. We show how these definitions correspond to logical terms: RDF graphs can be seen as logical formulae and RDF interpretations as logic interpretations. We show that our translation preserves models. Next we discuss a view on RDF which is motivated by research on databases: An RDF graph can also be modelled as an interpretation. In similarity to [8] we show that this view allows for two other characterisations of RDF entailment: as validity or as homomorphisms. In addiction we discuss a notion of minimal equivalent RDF graphs, called lean RDF graphs.

We next recall several aspects from logic programming. We relate logic programming to theorem proving. We argue that logic programming is a syntactical restriction of theorem proving which may be evaluated more efficiently than the general case. This is motivated by the fact that the language RDFLog is a syntactical extention of logic programming enjoying the computational properties of logic programming. We shall later show that this comes at the cost that models generated by RDFLog are not minimal. We finally recall the definition of the semantics of logic programming in terms of minimal Herbrand models and efficient algorithms to compute these.

RDFLog is introduced by some examples. We show that both RDFLog queries and answers can be modelled as RDF graphs. We show that the answer of a lean query to a lean RDF graph may be non lean. Nonetheless, if the queried RDF graph is lean, it holds under a closed world assumption that all blank nodes in the answer represent different entities. This motivates to deliver the proof for this fact to the user: for every blank node in the answer the concise bounded description may be reported to the user. This gives rise to the concise bounded answer. Next we show now blank nodes in the heads of RDFLog rules

may be used to construct new blank nodes and to group nodes in RDF graphs. Finally we define a Syntax for RDFLog.

In Chapter 6 we investigate how RDFLog may be evaluated operationally. The key idea is the reduction to logic programming: An RDFLog program is translated into a logic program by Skolemisation. The resulting program may be evaluated efficiently. Finally the result of the evaluation must be un-Skolemised to be a logical consequence of the program.

Finally we search for a suitable denotational characterisation of RDFLog. We show that the operational semantics S of an RDFLog program P has the following properties: S is entailed by P, it can be seen as a model of P and any RDF graph which is entailed by P is also entailed by S. We show how combinations of these properties can be used to characterise the operational semantics up to equivalence. We then turn to the problem of characterisation of the operational semantics up to equivalence. We provide such an characterisation but show that the implementation is DP-hard.

The reduction of RDF querying to logic programming can also be used to obtain complexity and expressivity results for RDF query languages. I particular it makes it easy to identify semantic restrictions of RDFLog with good computational properties. The main observation is that the construction of dependant blank nodes corresponds to function symbols in logic programming. Therefore there are two staight forward approaches to obtain decidable sublanguages of RDFLog: To disallow recursion or to disallow the construction of blank nodes. But these restrictions might be to strong. Much work has been done in the area of termination of logic programming [19, 4, 70]. It would surely be interesting to investigate which of these restrictions are well suited in the context of RDF querying.

Chapter 11

Outlook

Several questions around querying RDF with RDFLog remain open. In this chapter we discuss these open questions and possible future work.

If RDFLog is seen as a practical query language for RDF a more user friendly syntax for RDFLog would be necessary. As the perspective of this work is rather theoretical, and the aim of RDFLog is to provide a tool to investigate querying RDF, this topic is not addressed here.

As blank nodes have similarities to null values in databases [22] it would also be interesting to compare querying RDF with querying incomplete databases [49, 3]. It might be possible to reuse efficient algorithms developed in the context of incomplete information in databases to querying RDF. A comparison with modal logics [13, 29] and description logics [5, 47] could lead to decidable fragments of RDFLog.

A challenging task would be to build an efficient system which uses backward chaining to evaluate RDFLog. Skolemisation and un-Skolemisation could be integrated into the resolution. Especially the combination with tabling would be tempting. Lazy Skolemisation could also increase efficiency. Even more challenging is the task of implementing a evaluator supporting the lean semantics.

In Chapter 7 we provide an sufficient condition for the commutativity of the consequence operator and the lean operator. It is not known whether this condition is also necessary. A necessary condition would ensure that if this condition dose not hold then the application of these operators do not commutate anymore. Such a condition could be seen as optimal as in this case the (computationally expensive) lean operator would not be applied too often.

We have shown in Chapter 7 that there are examples where the semantics of an RDFLog program is infinite while the lean semantics is finite. Therefore it would be interesting to investigate whether the commutativity result of Chapter 7 has the property that any finite lean semantics can be computed in finite space. If this is not the case one could investigate whether there is some other condition or a fragment of RDFLog with this property.

Finally there are several unanswered questions around the definition of the semantics of RDFLog. Is the lean semantics of RDFLog a model of RDFLog? And is the minimal set of RDF graphs which entail every RDF graph, which is entailed by an RDFLog program isomorphic to the lean semantics? These questions are more of theoretical interest, but could help the understanding of RDFLog and RDF querying.

Acknowledgements.

This research has been funded by the European Commission and by the Swiss Federal Office for Education and Science within the 6th Framework Programme project REWERSE number 506779 (cf. `http://rewerse.net`).

Bibliography

[1] S. Abiteboul and V. Vianu. Datalog extensions for database queries and updates. *Journal of Computer and System Sciences*, 43(1):62–124, 1991.

[2] A. Alashqur and P. Apers. OQL: A Query Language for Manipulating Object-oriented Databases. *Proceedings of the Fifteenth International Conference on Very Large Databases*, pages 433–442, 1993.

[3] L. Antova, C. Koch, and D. Olteanu. World-set decompositions: Expressiveness and efficient algorithms. *Proc. ICDT*, pages 194–208, 2007.

[4] T. Arts and H. Zantema. Termination of logic programs using semantic unification. In *LOPSTR '95: Proceedings of the 5th International Workshop on Logic Programming Synthesis and Transformation*, pages 219–233, London, UK, 1996. Springer-Verlag.

[5] F. Baader, D. Calvanese, D. L. McGuinness, D. Nardi, and P. F. Patel-Schneider, editors. *The description logic handbook: theory, implementation, and applications*. Cambridge University Press, New York, NY, USA, 2003.

[6] L. Bachmair and H. Ganzinger. Rewrite-based Equational Theorem Proving with Selection and Simplification. *Journal of Logic and Computation*, 4(3):217, 1994.

[7] A. Badia and S. Vansummeren. Non-linear prefixes in query languages. In *PODS '07: Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 185–194, New York, NY, USA, 2007. ACM Press.

[8] J. Baget. RDF Entailment as a Graph Homomorphism. *Proceedings of ISWC*, 5, 2005.

[9] J. Bailey, F. Bry, T. Furche, and S. Schaflert. Web and semantic web query languages: A survey. *Lecture notes in computer science*, pages 35–133, 2005.

[10] D. Becket and J. Broekstra. SPARQL Query Results XML Format, 2006.

[11] D. Beckett. Turtle-Terse RDF Triple Language. *ILRT, University of Bristol*, 2004.

[12] T. Berners-Lee. Notation 3. *The World Wide Web Consortium(W3C) MIT, INRIA, http://www.w3.org/DesignIssues/Notation3.html. Design Suggestion*, 1998.

[13] P. Blackburn, M. de Rijke, and Y. Venema. *Modal logic*. Cambridge University Press New York, NY, USA, 2001.

[14] A. Blass and Y. Gurevich. Henkin quantifiers and complete problems. *Annals of Pure and Applied Logic*, 32:1–16, 1986.

[15] A. Bonner and M. Kifer. Transaction logic programming. *Proceedings of the Tenth International Conference on Logic Programming (ICLP)*, pages 257–279, 1993.

[16] A. Bonner and M. Kifer. An Overview of Transaction Logic. *Theoretical Computer Science*, 133(2):205–265, 1994.

[17] J. Broekstra and A. Kampman. SeRQL: A Second Generation RDF Query Language. *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*, 2003.

[18] J. Broekstra and A. Kampman. SeRQL: An RDF Query and Transformation Language. *Submitted to the International Semantic Web Conference, ISWC*, 2004.

[19] M. Bruynooghe, M. Codish, J. P. Gallagher, S. Genaim, and W. Vanhoof. Termination analysis of logic programs through combination of type-based norms. *ACM Trans. Program. Lang. Syst.*, 29(2):10, 2007.

[20] L. Cabibbo. The expressive power of stratified logic programs with value invention. *Information and Computation*, 147(1):22–56, 1998.

[21] J. Carrol and B. McBride. The Jena Semantic Web Toolkit. *Public api, HP-Labs, Bristol*, 2001.

[22] H. Chen, Z. Wu, H. Wang, and Y. Mao. Rdf/rdfs-based relational database integration. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 94, Washington, DC, USA, 2006. IEEE Computer Society.

[23] W. Chen, M. Kifer, and D. Warren. HiLog: A First-Order Semantics for Higher-Order Logic Programming Constructs. *Logic Programming: Proceedings of the North American Conference*, 2, 1989.

[24] W. Chen, M. Kifer, and D. Warren. HILOG: a foundation for higher-order logic programming. *Journal of Logic Programming*, 15(3):187–230, 1993.

[25] W. Clocksin and C. Mellish. Programming in Prolog. 1987.

[26] R. Cyganiak. A Relational Algebra for Sparql. 2005.

[27] S. Decker, M. Sintek, A. Billig, N. Henze, P. Dolog, W. Nejdl, A. Harth, A. Leicher, S. Busse, J. Ambite, et al. TRIPLE-an RDF Rule Language with Context and Use Cases. *Proceedings of W3C Workshop on Rule Languages for Interoperability,(Washington, DC, USA, April 2005), W3C*, pages 27–28, 2003.

[28] H. Ebbinghaus and J. Flum. *Mathematical Logic*. Springer, 1996.

[29] E. Emerson. Temporal and modal logic. *Handbook of Theoretical Computer Science*, pages 997–1072, 1990.

[30] R. Fagin, P. G. Kolaitis, and L. Popa. Data exchange: Getting to the core. *ACM Transactions on Database Systems*, 30(1):174–210, 2005.

[31] E. Franconi and S. Tessaris. The Sematics of SPARQL. Working Draft. *Lecture notes in computer science*, November 2005.

[32] T. Furche, B. Linse, F. Bry, D. Plexousakis, and G. Gottlob. RDF Querying: Language Constructs and Evaluation Methods Compared. *Reasoning Web*, 2006.

[33] K. Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik*, 37(1):349–360, 1930.

[34] G. Gottlob, N. Leone, and F. Scarcello. Hypertree Decompositions: A Survey. *Proceedings of the 26th International Symposium on Mathematical Foundations of Computer Science*, pages 37–57, 2001.

[35] C. Gutierrez, C. Hurtado, and A. Mendelzon. Foundations of semantic web databases. *Proceedings of the twenty-third ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 95–106, 2004.

[36] C. Gutiérrez, C. A. Hurtado, and A. O. Mendelzon. Formal aspects of querying rdf databases. In *Proc. Semantic Web and Databases Workshop (SWDB)*, pages 293–307, 2003.

[37] P. Haase, J. Broekstra, A. Eberhart, and R. VOLZ. A comparison of RDF query languages. *Lecture notes in computer science*, 10(1):502–517, 2004.

[38] S. Harris and N. Shadbolt. SPARQL query processing with conventional relational database systems. *Lecture notes in computer science*, pages 235–244, 2005.

[39] M. Harrison and M. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1978.

[40] P. Hayes. RDF Semantics, February 2004.

[41] L. Henkin. The Completeness of the First-Order Functional Calculus. *The Journal of Symbolic Logic*, 14(3):159–166, 1949.

[42] L. Henkin. Some remarks on infinitely long formulas. *Innitistic Methods*, pages 167–183, 1961.

[43] J. Herbrand. Investigations in proof theory. *From Frege to Godel: A Source Book in Mathematical Logic*, 1931:525–581, 1879.

[44] W. Hodges. *Model Theory*. Cambridge University Press, 1993.

[45] J. Hopcroft, R. Motwani, and J. Ullman. Introduction to automata theory, languages, and computation. *ACM SIGACT News*, 32(1):60–65, 2001.

[46] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 3rd edition, 2006.

[47] I. Horrocks, U. Sattler, and S. Tobies. Practical reasoning for very expressive description logics. *Logic Journal of IGPL*, 8(3):239, 2000.

[48] R. Hull and J. Su. Domain independence and the relational calculus. *Acta Informatica*, 31(6):513–524, 1994.

[49] T. Imieliński and J. Witold Lipski. Incomplete information in relational databases. *J. ACM*, 31(4):761–791, 1984.

[50] G. Karvounarakis, S. Alexaki, V. Christophides, D. Plexousakis, and M. Scholl. RQL: a declarative query language for RDF. *Proceedings of the eleventh international conference on World Wide Web*, pages 592–603, 2002.

[51] G. Karvounarakis, A. Magganaraki, S. Alexaki, V. Christophides, D. Plexousakis, M. Scholl, and K. Tolle. Querying the Semantic Web with RQL. *Computer Networks*, 42(5):617–640, 2003.

[52] M. Kifer and G. Lausen. F-logic: a higher-order language for reasoning about objects, inheritance, and scheme. *Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 134–146, 1989.

[53] G. Klyne and J. Carroll. Resource Description Framework (RDF): Concepts and Abstract Syntax, February 2004.

[54] O. Lassila, R. Swick, et al. Resource Description Framework (RDF) Model and Syntax Specification. 1999.

[55] J. Lloyd. *Foundations of logic programming*. Springer-Verlag New York, Inc. New York, NY, USA, 1987.

[56] D. Maier and D. Warren. *Computing with logic: logic programming with Prolog*. Benjamin-Cummings Publishing Co., Inc. Redwood City, CA, USA, 1988.

[57] F. Manola, E. Miller, et al. RDF Primer. *W3C Recommendation*, 10, 2004.

[58] A. Martelli and U. Montanari. Unification in Linear Time and Space. *Nota Interna B*, 76, 1976.

[59] A. Martelli and U. Montanari. An Efficient Unification Algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(2):258–282, 1982.

[60] W. McCune and R. Padmanabhan. *Automated Deduction in Equational Logic and Cubic Curves*. Springer, 1996.

[61] L. Miller, A. Seaborne, and A. Reggiori. Three Implementations of SquishQL, a Simple RDF Query Language. *Proceedings of the First International Semantic Web Conference on The Semantic Web*, pages 423–435, 2002.

[62] R. O'Keefe. *The craft of Prolog*. MIT Press Cambridge, MA, USA, 1990.

[63] J. Perez, M. Arenas, and C. Gutierrez. Semantics and complexity of sparql. In *Proc. Int'l. Semantic Web Conf. (ISWC)*, 2006.

[64] J. Perez, M. Arenas, and C. Gutierrez. Semantics and Complexity of SPARQL. *Arxiv preprint cs/0605124*, 2006.

[65] E. Prud'hommeaux and A. Seaborne. Sparql query language for rdf (working draft). March 2007.

[66] W. Quine. Epistemology Naturalised. *Ontological Relativity and Other Essays*, pages 69–90, 1969.

[67] W. Quine. *Mathematical Logic*. Harvard University Press, 1981.

[68] A. Riazanov. *Implementing an Efficient Theorem Prover*. PhD thesis, the University of Manchester, 2003.

[69] J. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*.

[70] D. D. Schreye and S. Decorte. Termination of logic programs: The never-ending story. *J. Log. Program.*, 19/20:199–260, 1994.

[71] A. Seaborne. RDQL-A Query Language for RDF. *W3C Member Submission*, 9, 2004.

[72] M. Sintek and S. Decker. TRIPLE-A Query, Inference, and Transformation Language for the Semantic Web. *International Semantic Web Conference (ISWC)*, pages 364–379, 2002.

[73] L. Sterling and E. Shapiro. The art of Prolog: advanced programming techniques. *Mit Press Series In Logic Programming*, page 427, 1986.

[74] P. Stickler. CBD – Concise Bounded Descriptions. *Online only, http://www.w3.org/Submission/CBD/*, 2004.

[75] H. Tamaki and T. Sato. OLDT Resolution with Tablulation. In *Proc. Int'l. Conf. on Logic Programming (ICLP)*, pages 84–98, 1986.

[76] G. Widmer and M. Kubat. Learning flexible concepts from streams of examples: FLORA2. *Proceedings of the 10th European conference on Artificial intelligence table of contents*, pages 463–467, 1992.

[77] G. Yang and M. Kifer. FLORA: Implementing an Efficient DOOD System Using a Tabling Logic Engine. *Lecture Notes in Computer Science*, 1861:1078, 2000.

[78] G. Yang and M. Kifer. Flora-2: User's Manual. *Department of Computer Sciene, Stony Brook University, Stony Brook*, 2002.

[79] G. Yang and M. Kifer. Inheritance and rules in object-oriented Semantic Web languages. *Lecture notes in computer science*, pages 95–110, 2003.

[80] G. Yang and M. Kifer. Reasoning about anonymous resources and meta statements on the semantic web. *Journal of Data Semantics*, 1:69–97, 2003.

[81] G. Yang, M. Kifer, and C. Zhao. FLORA-2: A rule-based knowledge representation and inference infrastructure for the Semantic Web. *Lecture notes in computer science*, pages 671–688, 2003.