An inference engine for RDF

Master thesis

G. Naudts

30 october 2003
Open University of the Netherlands
Agfa Gevaert

This document is the Master Thesis made as a part of my Master Degree in Computer Science Education (Software Systems) at the Open University of the Netherlands. The work has been done in collaboration with the research department of the company Agfa in Mortsel Belgium.

Student data

Name	Guido Naudts
Student number	831708907
Address	Secretarisdreef 5
	2288 Bouwel
Telephone work	0030-2-542.76.01
Home	0030-14-51.32.43
E-mail	Naudts_Vannoten@yahoo.com

Coaching and graduation committee

Chairman:	dr J.T. Jeuring, professor at the Open University
Secretary :	ir. F.J. Wester, senior lecturer at the Open University
Coach:	ir. J. De Roo, researcher at Agfa Gevaert.

Acknowledgements

I want to thank Ir. J. De Roo for giving me the opportunity for making a thesis about a tremendous subject and his guidance in the matter.
Prof. J. T. Jeuring and Ir. F. J. Wester are thanked for their efforts to help me produce a readable and valuable text.
I thank M. P. Jones for his Prolog demo in the Hugs distribution that has been very helpful.
I thank all the people from the OU for their efforts during the years without which this work would not have been possible.
I thank my wife and children for supporting a father seated behind his computer during many years.

Summary

The Semantic Web is an initiative of the *World Wide Web Consortium* (**W3C**). The goal of this project is to define a series of standards. These standards will create a framework for the automation of activities on the *World Wide Web* (**WWW**) that still need manual intervention by human beings at this moment.
A certain number of basic standards have been developed already. Among them **XML** is without doubt the most known. Less known is the *Resource Description Framework* (**RDF**). This is a standard for the creation of meta information. This is the information that has to be given to computers to permit them to handle tasks previously done by human beings.
Information alone however is not sufficient. In order to take a decision it is often necessary to follow a certain reasoning. Reasoning is connected with logics. The consequence is that computers have to be fed with programs and rules that can take decisions, following a certain logic, based on available information.
These programs and reasonings are executed by *inference engines*. The definition of inference engines for the Semantic Web is at the moment an active field of experimental research.
These engines have to use information that is expressed following the standard **RDF**. The consequences of this requirement for the definition of the engine are explored in this thesis.
An inference engine uses information and, given a *query*, reasons with this information with a *solution* as a result. It is important to be able to show that the solutions, obtained in this manner, have certain characteristics. These are, between others, the characteristics *soundness* and *completeness.* This proof is delivered in this thesis by means of a reasoning based on the graph theoretical properties of **RDF**.
On the other hand these engines must take into account the specificity of the World Wide Web. One of the properties of the web is the fact that sets are not closed. This implies that not all elements of a set are known. Reasoning then has to happen in an *open world.* This fact has far reaching logical implications.
During the twentieth century there has been an impetous development of logic systems. More than 2000 kinds of logic were developed. Notably *first order logic* (**FOL**) has been under heavy attack, beside others, by the dutch Brouwer with the *constructive* or *intuistionistic* logic. Research is also done into kinds of logic that are suitable for usage by machines.
In this thesis I argue that an inference engine for the World Wide Web should follow a kind of constructive logic and that **RDF**, with a logical *implication* added, satisfies this requirement. It is shown that a constructive approach is important for the verifiability of the results of inferencing.

A query on the World Wide Web can extend over more than one server. This means that a process of subquerying must be defined. This concept is introduced in an experimental inference engine, **RDFE**ngine.
An important characteristic for an engine that has to be active in the World Wide Web is *efficiency*. Therefore it is important to do research about the methods that can be used to make an efficient engine. The structure has to be such that it is possible to work with huge volumes of data. Eventually these data are kept in relational databases.
On the World Wide Web there is information available in a lot of places and in a lot of different forms. Joining parts of this information and reasoning about the result can easily lead to the existence of *contradictions* and *inconsistencies*. These are inherent to the used logic or are dependable on the application. A special place is taken by inconsistencies that are inherent to the used ontology. For the Semantic Web the ontology is determined by the standards *rdfs* and *OWL*.
An ontology introduces a classification of data and applies restrictions to those data. An inference engine for the Semantic Web needs to have such characteristics that it is compatible with rdfs and **OWL**.
An executable specification of an inference engine in Haskell was constructed. This permitted to test different aspects of inferencing and the logic connected to it. This engine has the name **RDFE**ngine. A large number of existing testcases was executed with the engine. A certain number of testcases was constructed, some of them for inspecting the logic characteristics of **RDF**.

Samenvatting

Het Semantisch Web is een initiatief van het *World Wide Web Consortium* (**W3C**). Dit project beoogt het uitbouwen van een reeks van standaarden. Deze standaarden zullen een framework scheppen voor de automatisering van activiteiten op het *World Wide Web* (**WWW**) die thans nog manuele tussenkomst vereisen.
Een bepaald aantal basisstandaarden zijn reeds ontwikkeld waaronder **XML** ongetwijfeld de meest gekende is. Iets minder bekend is het *Resource Description Framework* (**RDF**). Dit is een standaard voor het creëren van meta-informatie. Dit is de informatie die aan de machines moet verschaft worden om hen toe te laten de taken van de mens over te nemen.
Informatie alleen is echter niet voldoende. Om een beslissing te kunnen nemen moet dikwijls een bepaalde redenering gevolgd worden. Redeneren heeft te maken met logica. Het gevolg is dat computers gevoed moeten worden met programma's en regels die, volgens een bepaalde logica, besluiten kunnen nemen aan de hand van beschikbare informatie.
Deze programma's en redeneringen worden uitgevoerd door zogenaamde *inference engines*. Dit is op het huidig ogenblik een actief terrein van experimenteel onderzoek.
De engines moeten gebruik maken van de informatie die volgens de standaard **RDF** wordt weergegeven. De gevolgen hiervan voor de structuur van de engine worden in deze thesis nader onderzocht.
Een inference engine gebruikt informatie en, aan de hand van een *vraagstelling*, worden redeneringen doorgevoerd op deze informatie met een *oplossing* als resultaat. Het is belangrijk te kunnen aantonen dat de oplossingen die aldus bekomen worden bepaalde eigenschappen bezitten. Dit zijn onder meer de eigenschappen *juistheid* en *volledigheid*. Dit bewijs wordt geleverd bij middel van een redenering gebaseerd op de graaf theoretische eigenschappen van **RDF**.
Anderzijds dienen de engines rekening te houden met de specifieke eigenschappen van het World Wide Web. Een van de eigenschappen van het web is het open zijn van verzamelingen. Dit houdt in dat van een verzameling niet alle elementen gekend zijn. Het redeneren dient dan te gebeuren in een *open wereld.* Dit heeft verstrekkende logische implicaties.
In de loop van de twintigste eeuw heeft de logica een stormachtige ontwikkeling gekend. Meer dan 2000 soorten logica werden ontwikkeld. Aanvallen werden doorgevoerd op het bolwerk van de *first order logica* (**FOL**), onder meer door de nederlander Brouwer met de *constructieve* of *intuistionistische* logica. Gezocht wordt ook naar soorten logica die geschikt zijn voor het gebruik door machines.
In deze thesis wordt betoogd dat een inference engine voor het **WWW** enerzijds een constructieve logica dient te volgen en anderzijds, dat **RDF** aangevuld met een logische *implicatie* aan dit soort logica voldoet. Het wordt aangetoond dat

een constructieve benadering belangrijk is voor de verifieerbaarheid van de resultaten van de inferencing.
Een query op het World Wide Web kan zich uitstrekken over meerdere servers. Dit houdt in dat een proces van subquerying moet gedefinieerd worden. Dit concept wordt geïntroduceerd in een experimentele inference engine, **RDFE**ngine.

Een belangrijke eigenschap voor een engine die actief dient te zijn in het World Wide Web is *efficiëntie*. Het is daarom belangrijk te onderzoeken op welke wijze een efficiënte engine kan gemaakt worden. De structuur dient zodanig te zijn dat het mogelijk is om met grote volumes aan data te werken, die eventueel in relationele databases opgeslagen zijn.
Op het World Wide Web is informatie op vele plaatsen en in allerlei vormen aanwezig. Het samenvoegen van deze informatie en het houden van redeneringen die erop betrekking hebben kan gemakkelijk leiden tot het ontstaan van *contradicties* en *inconsistenties.* Deze zijn inherent aan de gebruikte logica of zijn afhankelijk van de applicatie. Een speciale plaats nemen de inconsistenties in die inherent zijn aan de gebruikt ontologie. Voor het Semantisch Web wordt de ontologie bepaald door de standaarden *rdfs* en *OWL*. Een ontologie voert een classificatie in van gegevens en legt ook beperkingen aan deze gegevens op. Een inference engine voor het Semantisch Web dient zulke eigenschappen te bezitten dat hij compatibel is met rdfs en **OWL**.
Een uitvoerbare specificatie van een inference engine in Haskell werd gemaakt. Dit liet toe om allerlei aspecten van inferencing en de ermee verbonden logica te testen. De engine werd RDFEngine gedoopt. Een groot aantal bestaande testcases werd onderzocht. Een aantal testcases werd bijgemaakt, sommige speciaal met het oogmerk om de logische eigenschappen van **RDF** te onderzoeken.

Chapter 1. Introduction

1.1. Overview

In chapter 1 a case study is given and then the goals of the research that is the subject of the thesis are exposed with the case study as illustration. The methods used are explained and an overview is given of the fundamental knowledge upon which the research is based. The relation with current research is indicated. Then the structure of the thesis is outlined.

1.2. Case study

1.2.1. Introduction

A case study can serve as a guidance to what we want to achieve with the Semantic Web. Whenever standards are approved they should be such that important case studies remain possible to implement. Discussing all possible application fields here would be out of scope. However one case study will help to clarify the goals of the research.

1.2.2. The case study

A travel agent in Antwerp has a client who wants to go to St.Tropez in France. There are rather a lot of possibilities for composing such a voyage. The client can take the train to France, or he can take a bus or train to Brussels and then the airplane to Nice in France, or the train to Paris then the airplane or another train to Nice. The travel agent explains the client that there are a lot of possibilities. During his explanation he gets an impression of what the client really wants. Fig.1.1. gives a schematic view of the case study.

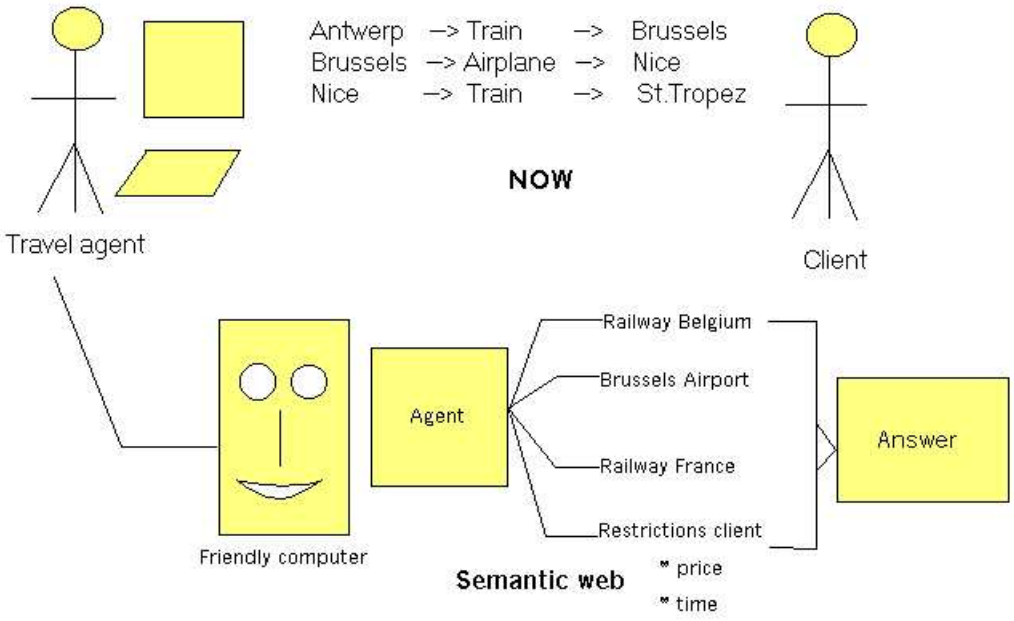

Fig.1.1. A Semantic Web case study

The travel agent agrees with the client about the itinerary: by train from Antwerp to Brussels, by airplane from Brussels to Nice and by train from Nice to St. Tropez. This still leaves room for some alternatives. The client will come back to make a final decision once the travel agent has said him by mail that he has worked out some alternative solutions like price for first class vs second class etc...
Remark that the decision for the itinerary that has been taken is not very well founded; only very crude price comparisons have been done based on some internet sites that the travel agent consulted during his conversation with the client. A very cheap flight from Antwerp to Cannes has escaped the attention of the travel agent.
The travel agent will now further consult the internet sites of the Belgium railways, the Brussels airport and the France railways to get some alternative prices, departure times and total travel times.

Now let's compare this with the hypothetical situation that a full blown Semantic Web would exist. In the computer of the travel agent resides a Semantic Web agent that has at its disposal all the necessary standard tools. The travel agent has a specialised interface to the general Semantic Web agent. He fills in a query in his specialised screen. This query is translated to a standardised query format for the Semantic Web agent. The agent consult his rule database. This database of course contains a lot of rules about travelling as well as facts like e.g. facts about internet sites where information can be

obtained. There are a lot of 'path' rules: rules for composing an itinerary (for an example of what such rules could look like see: [GRAPH]. The agent contacts different other agents like the agent of the Belgium railways, the agents of the French railways, the agent of the airports of Antwerp, Brussels, Paris, Cannes, Nice etc...
With the information received its inference rules about scheduling a trip are consulted. This is all done while the travel agent is chatting with the client to detect his preferences. After some 5 minutes the Semantic Web agent gives the travel agent a list of alternatives for the trip; now the travel agent can immediately discuss this with his client. When a decision has been reached, the travel agent immediately gives his Semantic Web agent the order for making the reservations and ordering the tickets. Now the client only will have to come back once for getting his tickets and not twice. The travel agent not only has been able to propose a cheaper trip as in the case above but has also gained an important amount of time.

1.2.3. Conclusions of the case study

That a realisation of a similar system is interesting is evident. Clearly, the standard tools do have to be very flexible and powerful to be able to put into rules the reasoning of this case study (path determination, itinerary scheduling). All these rules then have to be made by someone. This can of course be a common effort for a lot of travel agencies.
What exists now? A quick survey learns that there are web portals where a client can make reservations (for hotel rooms). However the portal has to be fed with data by the travel agent. There also exist software that permits the client to manage his travel needs. But all that software has to be fed with information obtained by a variety of means, practically always manually.
A partial simulation namely the order of a ticket for a flight can be found in 5.16.5.

1.3. Research goals

1.3.1. Standards

In the case study different information sites have to communicate with another. If this has to be done in an automatic way a standard has to be used. The standard that is used for expressing information is **RDF**. **RDF** is a standard for expressing meta-information developed by the **W3C**. The primary purpose is adding meta-information to web pages so that these become more usable for computers [TBL]. Later the Semantic Web initiative was launched by Berners-Lee with the purpose of developing standards for automating the exchange of

information between computers. The information standard is again **RDF** [DESIGN]. However other standards are added on top of **RDF** for the realisation of the Semantic Web (chapter 2).

1.3.2. Research questions

The following research questions were defined:

1) *define an inference process on top of RDF and give a specification of this process in a functional language.*

   In the case study the servers dispose of information files that contain facts and rules. The facts are expressed in **RDF**. The syntax of the rules is **RDF** also, but the semantic interpretation is different. Using rules implies inferencing. On top of **RDF** a standardized inference layer has to be defined. A functional language is very well suited for a declarative specification. At certain points the inference process has to be interrupted to direct queries to other sites on the World Wide Web. This implies a process of *subquerying* where the inference process does not take place in one site (one computer) but is distributed over several sites (see also fig. 1.1).

2) *which kind of logic is best suited for the Semantic Web?*

   Using rules; making queries; finding solutions; this is the subject of logic. So the relevance of logic for an inferencing standard based on top of **RDF** has to be investigated.

3) *is the specified inference process sound and complete?*

   It is without question of course that the answers to a query have to be valid. By completeness is meant that all answers to a query are found. It is not always necessary that all answers are found but often it is.

4) *what can be done for augmenting the efficiency of the inference process?*

   A basic internet engine should be fast because the amount of data can be high; it is possible that data need to be collected from several sites over the internet. This will already imply an accumulation of transmission times. A basic engine should be optimized as much as possible.

5) *how can inconsistencies be handled by the inference process?*

> In a closed world (reasoning on one computer system) inconsistencies can be mastered; on the internet, when inferencing becomes a distributed process, inconsistencies will be commonplace.

As of yet no standard is defined for inferencing on top of **RDF**. As will be shown in this thesis there are a lot of issues involved and a lot of choices to be made. The definition of such a standard is a necessary, but difficult step in the progress of the Semantic Web. In order to be able to define a standard a definition has to be provided for following notions that represent minimal extensions on top of **RDF**: rules, queries, solutions and proofs.

1.4. Research methods

The primary research method consists of writing an implementation of an **RDF** inference engine in Haskell. The engine is given the name **RDFE**ngine. The executable specification is tested using test cases where the collection of test cases of De Roo [DEROO] plays an important role. Elements of logic, efficiency and inconsistency can be tested by writing special test cases in interaction with a study of the relevant literature.

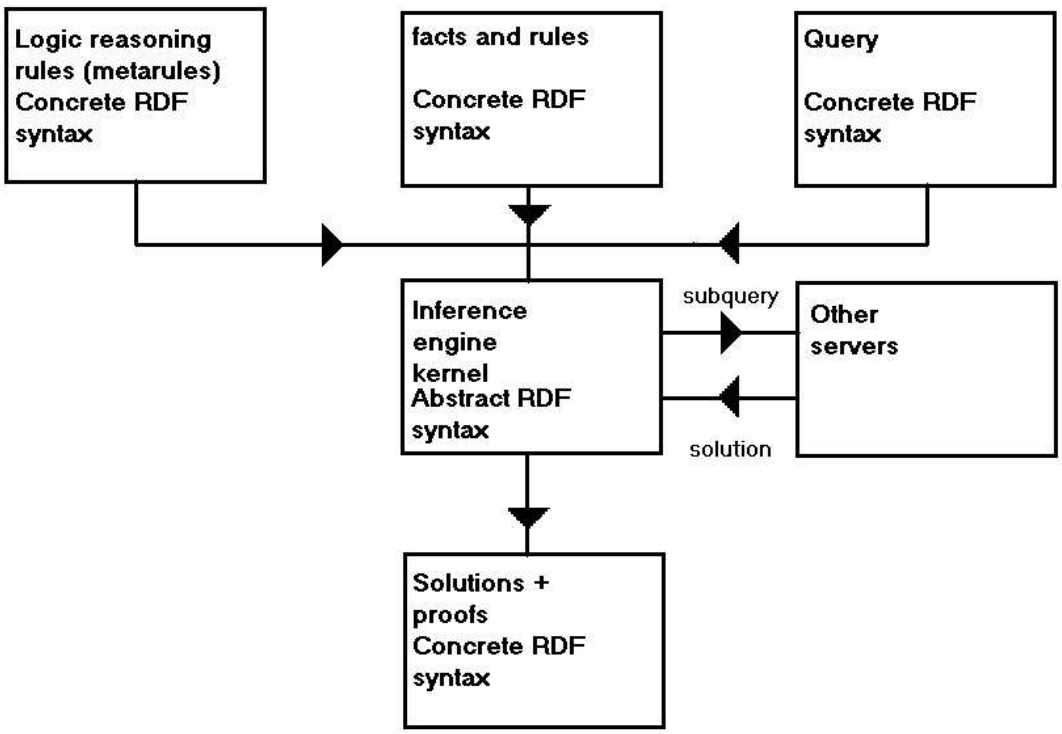

Fig.1.2. The inference engine **RDFE**ngine in a server with its inputs and outputs.

In fig.1.2 the global structure of the inference engine is drawn. The inputs and outputs are in a concrete **RDF** syntax. Notation 3 (N3) and **RDFP**rolog are used

by **RDFE**ngine. **RDFP**rolog was developed with no special purpose, but often I found it more convenient to use than Notation 3.
The engine is given a modular structure and different versions exist for different testing purposes.
For making a Haskell implementation the choice was made for a resolution based engine. A forward reasoning engine could have been chosen as well but a resolution based engine is more goal directed and has better characteristics for subquerying.
Other used methods are:

- the Haskell specification describes the abstract syntax and the data structures used by **RDFE**ngine.
- the study of aspects of inferencing that are specific to the World Wide Web. These aspects include the problems related to the notions open/closed world, subquerying, verifiability of results and inconsistencies.
- the graph description of the inferencing process

1.5. The basic material

The research used amongst others the following material:

a) the collection of test cases of De Roo [DEROO]
b) the **RDF** model theory [RDFM] [RDFMS]
c) the **RDF** rules mail list [RDFRULES]
d) the classic resolution theory [VANBENTHEM] [STANFORD]
e) logic theories and mainly first order logic, constructive logic and paraconsistent logic. [VANBENTHEM, STANFORDP]

1.6. Related work

The mail list **RDF** rules [RDFRULES] has as a purpose to discuss the extension of **RDF** with inferencing features, mainly rules, queries and answers. There are a lot of interesting discussions but no clear picture is emerging at the moment.
Berners-Lee [DESIGN] seeks inspiration in first order logic. He defines logic predicates (chapter 2), builtins, test cases and a reference engine **CWM**. However the semantics corresponding to the input and output formats for a **RDF** engine are not clearly defined.
A common approach is to use existing logic systems. **RDF** is then expressed in terms of these systems and logic (mostly rules) is added on top e.g. Horn logic or frame logic. An example is found in [DECKER]. In this case the semantics are those of the used logic.
The implementation of an ontology like **OWL** (Ontology Web Language) implies also inferencing facilities that are necessary for implementing the semantics of the ontology [OWL features].

Euler [DEROO] and **CWM** take a graph oriented approach for the construction of the inference program. The semantics are mostly defined by means of test cases. The approach taken by my research for defining the semantics of the inference input and output has two aspects:

1) it is graph oriented and, in fact, the theory is generally applicable to labeled graphs.
2) from the logic viewpoint a constructive approach is taken for reasons of verifiability (chapter 4).
3) a constructive graph oriented definition is given for rules, queries, solutions and proofs.

1.7. Outline of the thesis

Chapter two gives an introduction into the basic standards that are used by **RDF** and into the syntax and model theory of **RDF**.

Chapter three discusses related work. It consists of three parts: automated reasoning, logic and existing software systems.

Chapter four explores in depth the relationship of **RDF** inferencing with logic. Here the accent is put on a constructive logic interpretation.

Chapter five exposes the meaning of inferencing using an **RDF** facts database. The Haskell specification, **RDFE**ngine, is explained and a theory of resolution based on reasoning on a graph is presented. Some applications are reviewed.

Chapter six discusses optimalization techniques. Especially important is a technique based on the theory exposed in chapter five.

Chapter seven disusses inconsistencies that can arise during the course of inferencing processes on the Semantic Web.

Chapter eight finally gives a conclusion and indicates possible ways for further research.

Chapter 2. Preliminaries

2.1. Introduction

In this chapter a number of techniques are briefly explained. These techniques are necessary building blocks for the Semantic Web. They are also a necessary basis for inferencing on the web.

2.2. XML and namespaces

2.2.1. Definition

XML (Extensible Markup Language) is a subset of **SGML** (Standard General Markup Language). In its original definition a markup language is a language which is intended for adding information ("markup" information) to an existing document. This information must stay separate from the original hence the presence of separation characters. In **SGML** and **XML** 'tags' are used.

2.2.2. Features

There are two kinds of tags: opening and closing tags. The opening tags are keywords enclosed between the signs "<" and ">". An example: <author>. A closing tag is practically the same, only the sign "/" is added e.g. </author>. With these elements alone very interesting datastructures can be built. An example of a book description:

```
<book>
  <title>
      The Semantic Web
  </title>
  <author>
      Tim Berners-Lee
  </author>
</book>
```

As can be seen it is quite easy to build hierarchical datastructures with these elements alone. A tag can have content too: in the example the strings "The Semantic Web" and "Tim Berners-Lee" are content. One of the good characteristics of **XML** is its simplicity and the ease with which parsers and other tools can be built.

The keywords in the tags can have attributes too. The previous example could be written:

<book title="The Semantic Web" author="Tim Berners-Lee"></book>

where attributes are used instead of tags.

XML is not a language but a meta-language i.e. a language with as goal to make other languages ("markup" languages).
Everybody can make his own language using **XML**. A person doing this only has to use the syntaxis of **XML** i.e. produce wellformed **XML**. However more constraints can be added to an **XML** language by using a **DTD** or an **XML** schema. A valid **XML** document is one that uses **XML** syntax and respects the constraints laid down in a **DTD** or **XML** schema.
If everybody creates his own language then the "tower-of-Babylon"-syndrome is looming. How is such a diversity in languages handled? This is done by using *namespaces*. A namespace is a reference to the definition of an XML language. Suppose someone has made an **XML** language about birds. Then he could make the following namespace declaration in **XML**:

<birds:wing xmlns:birds="http://birdSite.com/birds/">

This statement is referring to the tag "wing" whose description is to be found on the site that is indicated by the namespace declaration xmlns (= **XML** namespace). Now our hypothetical biologist might want to use an aspect of the physiology of birds described however in another namespace:

<physiology:temperature xmlns:physiology=" http://physiology.com/xml/">

By the semantic definition of **XML** these two namespaces may be used within the same **XML**-object.

```
<?xml version="1.0" ?>
<birds:wing xmlns:birds="http://birdSite.com/birds/">
        large
</birds:wing>
<physiology:temperature xmlns:physiology=" http://physiology.com/xml/">
        43
</physiology:temperature>
```

2.3. URI's and URL's

2.3.1. Definitions

URI stands for **U**niform **R**esource **I**ndicator. A **URI** is anything that indicates unequivocally a resource.
URL stands for **U**niform **R**esource **L**ocator. This is a subset of **URI**. A **URL** indicates the access to a resource. **URN** refers to a subset of **URI** and indicates names that must remain unique even when the resource ceases to be available.
URN stands for **U**niform **R**esource **N**ame. [ADDRESSING]
In this thesis only **URL**'s will be used.

2.3.2. Features

The following example illustrates a **URL** format that is in common use. [URI]:

http:// www.math.uio.no/faq/compression-faq/part1.html

Most people will recognize this as an internet address. It unequivocally identifies a web page. Another example of a **URI** is a **ISBN** number that unquivocally identifies a book.

The general format of an http **URL** is:

http://<host>:<port>/<path>?<searchpart>.

The host is of course the computer that contains the resource; the default port number is normally 80; eventually e.g. for security reasons it might be changed to something else; the *path* indicates the directory access path. The *searchpart* serves to pass information to a server e.g. data destinated for **CGI-**scripts.
When an **URL** finishes with a slash like *http://www.test.org/definitions/* the directory *definitions* is addressed. This will be the directory defined by adding the standard prefix path e.g. */home/netscape* to the directory name: */home/netscape/definitions*.The parser can then return e.g. the contents of the directory or a message 'no access' or perhaps the contents of a file 'index.html' in that directory.
A path might include the sign "#" indicating a named anchor in an html-document. Following is the html definition of a named anchor:

<H2><A NAME="semantic">The Semantic Web</A></H2>

A named anchor thus indicates a location within a document. The named anchor can be called e.g. by:

http://www.test.org/definition/semantic.html#semantic

2.4. Resource Description Framework RDF

2.4.1. Introduction

RDF is a language. The semantics are defined by [RDFM]; some concrete syntaxes are: **XML**-syntax, Notation 3, N-triples, **RDFP**rolog. N-triples is a subset of Notation 3 and thus of **RDF** [RDFPRIMER]. **RDFP**rolog is also a subset of **RDF** and is defined in this thesis.
Very basically **RDF** uses triples: *(subject, predicate, object)*. The predicate is called *the property*. This simple statement however is not the whole story; nevertheless it is a good point to start.
An example [ALBANY] of a statement is:
"Jan Hanford created the J. S. Bach homepage.". The J.S. Bach homepage is a resource. This resource has a **URI**: *http://www.jsbach.org/*. It has a property: creator with value 'Jan Hanford'. Figure 2.1. gives a graphical view of this.

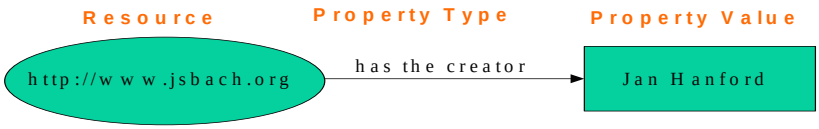

Fig.2.1. An example of a **RDF** relation.

In simplified **RDF** this is:

```
<RDF>
  <Description about= "http://www.jsbach.org">
    <Creator>Jan Hanford</Creator>
  </Description>
</RDF>
```

However this is without namespaces meaning that the notions are not well defined. With namespaces added this becomes:

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:dc="http://purl.org/DC/">
  <rdf:Description about="http://www.jsbach.org/">
    <dc:Creator>Jan Hanford</dc:Creator>
  </rdf:Description>
</rdf:RDF>
```

The first namespace *xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"* refers to the document describing the (**XML**-)syntax of **RDF**; the second namespace *xmlns:dc="http://purl.org/DC/*" refers to the description of the Dublin Core, a basic ontology about authors and publications. This is also an example of two languages that are mixed within an **XML**-object: the **RDF** and the Dublin Core language.

In the following example is shown that more than one predicate-value pair can be indicated for a resource.

```
<rdf:RDF xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:bi="http://www.birds.org/definitions/">
  <rdf:Description about= "http://www.birds.org/birds#swallow">
    <bi:wing>pointed</bi:wing>
    <bi:habitat>forest</bi:habitat>
  </rdf:Description>
</rdf:RDF>
```

2.4.2. Verifiability of a triple

RDF triples can have different origins. When site *A* recieves a triple from site *D: (B:s, C:p, C:o)* where *B* and *C* are namespaces, then *A* may wish to verify the trustworthiness of the labels *B:s*, *C:p* and *C:o.*
Consider the subject of the triple *B:s*. Things that can be verified are:

a) the existence of the namespace *B.*
b) whether the site or owner of namespace *B* is trusted.
c) the notion associated with *s* is defined in namespace *B.*
d) eventually the triple might be signed and the signature of the triple can be verified.

2.4.3. The graph syntax

In the graph representation subjects and objects are *nodes* in the graph and the *arcs* represent properties. No nodes can occur twice in the graph. The consequence is that some triples will be connected with other triples; other

triples will have no connections. Also the graph will have different connected subgraphs.
A node can also be an *blank* node which is a kind of anonymous node. A blank node can be *instantiated* with a URI or a literal and then becomes a 'normal' node.

2.4.4. Conclusion

What is **RDF**? It is a language with a simple semantics consisting of triples: (subject, predicate, object) and some other elements. Several syntaxes exist for **RDF**: **XML**, graph, Notation 3, N-triples, **RDFP**rolog. Notwithstanding its simple structure a great deal of information can be expressed with it.

2.5. Notation 3

2.5.1. Introduction

Notation 3 is a syntax for RDF developed by Tim Berners-Lee and Dan Connolly and represents a more human usable form of the RDF-syntax with in principle the same semantics [RDF Primer]. The first version of **RDFE**ngine uses Notation 3. The second version uses **RDFP**rolog (see 2.8). An alternative and shorter name for Notation 3 is N3.

2.5.2. Basic properties

The basic syntactical form in Notation 3 is a *triple* of the form :
<*subject*> <*property*> <*object*> where subject, property and object are atoms. An atom can be either a **URI** (or a **URI** abbreviation), a blank node or a variable[1]. Subject and object can also be triplelists. There also is some " syntactic sugar".

In N3 **URI**'s can be indicated in a variety of different ways. A **URI** can be indicated by its complete form or by an abbreviation. These abbreviations have practical importance and are intensively used in the testcases. Here I give the most important features; in appendix 5 a more complete overview is given.
A *@prefix* instruction defines a namespace abbreviation. All namespace abbreviations thus defined end with the character ':'. Example:

@prefix ont: <http://www.daml.org/2001/03/daml-ont#>.

Here the namespace *http://www.daml.org/2001/03/daml-ont#* is abbreviated to *ont:*
For example,

<http://www.daml.org/2001/03/daml-ont#TransitiveProperty> .

[1] Variables are an extension to RDF.

is abbreviated to *ont:TransitiveProperty*.
by the prefix instruction above.
The signs <> or <#> indicate the **URI** of the current document i.e. the document that contains the Notation 3 file.

Two substantial abbreviations for *sets of triples* are property lists and object lists. It can happen that a subject recieves a series of qualifications; each qualification with a verb and an object,
e.g. *:bird :color :blue; height :high; :presence :rare.*
This represents a bird with a blue color, a high height and a rare presence.
These properties are separated by a semicolon.
A verb or property can have several values e.g.
:bird :color :blue, :yellow, : black.
This means that the bird has three colors. This is called an object list. The two things can be combined :

:bird :color :blue, :yellow, : black; height :high; presence :rare.

The objects in an objectlist are separated by a comma.
A semantic and syntactic feature are anonymous subjects. The symbols '[' and ']' are used for this feature. [:can :swim]. means there exists an anonymous subject x that can swim. The abbreviations for propertylist and objectlist can here be used too :

[:can :swim, :fly; :color :yellow].

The property rdf:type can be abbreviated as "a":

:lion a :mammal.

really means:

:lion rdf:type :mammal.

2.6. The logic layer

In [SWAPLOG] an experimental logic layer is defined for the Semantic Web.(I will say a lot more about logic in chapter 4.) This layer is defined using Notation 3. An overview of the most important features (**RDFE**ngine only uses log:implies, log:forAll, log:forSome and log:Truth):

log:implies : this is the implication, also indicated by an arrow: →.

Example:
{:rat a :rodentia. :rodentia a :mammal.}→{:rat a :mammal}.

log:forAll : the purpose is to indicate universal variables :

Example:

{:x rdfs:subClassOf :y. :y rdfs:subClassOf :z. rdfs:subClassOf a owl:transitiveProperty}→{:x rdfs:subClassOf :z.}; log:forAll :x, :y, :z.

indicates that *:x, :y* and *:z* are universal variables.

log:forSome: does the same for existential variables:

Example:

{:x a :human.}→{:x a :man}; log:forSome :x.

log:Truth : states that this is a universal truth. This is not interpreted by **RDFE**ngine.

Here follow briefly some other features:
log:falseHood : to indicate that a formula is not true.
log:conjunction : to indicate the conjunction of two formulas.
log:includes : *F* includes *G* means *G* follows from *F*.
log:notIncludes: *F* notIncludes *G* means *G* does not follow from *F*.

2.7. Semantics of N3

2.7.1. Introduction

The semantics of N3 are the same as the semantics of **RDF** [RDFM].
The semantics indicate the correspondence between the syntactic forms of **RDF** and the entities in the universum of discourse. The semantics are expressed by a model theory. A *valid triple* is a triple whose elements are defined by the model theory. A *valid RDF graph* is a set of valid RDF triples.

2.7.2. The model theory of RDF

The vocabulary *V* of the model is composed of a set of **URI**'s.
LV is the set of *literal values* and *XL* is the mapping from the literals to *LV*.

A *simple interpretation I* of a vocabulary *V* is defined by:

1. a non-empty set *IR* of resources, called the domain or universe of *I*.

2. a mapping *IEXT* from *IR* into the powerset of *IR X (IR union LV)* i.e. the set of sets of pairs <*x,y*> with *x* in *IR* and *y* in *IR* or *LV*. This mapping defines the properties of the **RDF** triples.

3. a mapping *IS* from *V* into *IR*

IEXT(x) is a set of pairs which identify the arguments for which the property is true, i.e. a binary relational extension, called the *extension* of x. [RDFMS]

Informally this means that every **URI** represents a resource which might be a page on the internet but not necessarily: it might as well be a physical object. A property is a relation; this relation is defined by an extension mapping from the property into a set. This set contains pairs where the first element of a pair represents the subject of a triple and the second element of a pair represent the object of a triple. With this system of extension mapping the property can be part of its own extension without causing paradoxes. This is explained in [RDFMS].

2.7.3. Examples

Take the triple:

:bird :color :yellow.

In the set of **URI**'s there will be things like: :bird, :mammal, :color, :weight, :yellow, :blue etc...These are part of the vocabulary *V*.

In the set **IR** of resources will be: #bird, #color etc.. i.e. resources on the internet or elsewhere. #bird might represent e.g. the set of all birds.

There then is a mapping *IEXT* from #color (resources are abbreviated) to the set {(#bird,#blue),(#bird,#yellow),(#sun,#yellow),...}

and the mapping *IS* from *V* to *IR*:

:bird → #bird, :color → #color, ...

The **URI** refers to a page on the internet where the domain *IR* is defined (and thus the semantic interpretation of the **URI**).

2.8. RDFProlog

2.8.1. Introduction

I have defined a language called **RDFP**rolog (by the Haskell module **RDFP**rolog.hs: for a sample see 5.17. Applications). This language is a concrete syntax with **RDF** semantics. It forms a subset of **RDF** and the syntax is less

complex than the Notation 3 syntax. The last version of **RDFE**ngine uses this syntax for input and output of **RDF** data.
This language is very similar to the language **DATALOG** that is used for a deductive access to relational databases [ALVES].
The parser is taken from [JONES] with minor modifications.

2.8.2. Syntax

General format of a triple:
predicate(subject, object).
Example: *name(company,"Test")* is a **RDF** triple where *name* is the predicate, *company* is the subject and *"Test"* is the object.
In **RDFP**rolog the general format of a rule is:

predicate-1(subject-1,object-1),…, predicate-n(subject-n,object-n) :> predicate-q(subject-q,object-q), …,predicate-z(subject-z,object-z).

where all predicates,subjects and objects can be variables.
The format of a fact is:

predicate-1(subject-1,object-1),…, predicate-n(subject-n,object-n).

where all predicates,subjects and objects can be variables.
Variables begin with capital letters.
Example:
child(X,Y), child(Y,Z) :> grandparent(X,Z).

A rule and a fact are to be entered on one line (no continuation). A line with a syntax error is neglected. This can be used to add comment.

2.9. A global view of the Semantic Web

2.9.1. Introduction.

The Semantic Web will be composed of a series of standards. These standards have to be organised into a certain structure that is an expression of their interrelationships. A suitable structure is a hierarchical, layered structure.

2.9.2. The layers of the Semantic Web

Fig.2.2. illustrates the different parts of the Semantic Web in the vision of Tim Berners-Lee. The notions are explained in an elementary manner here. Later some of them will be treated more in depth.

Layer 1. Unicode and **URI**

At the bottom there is Unicode and **URI**. Unicode is the Universal code.

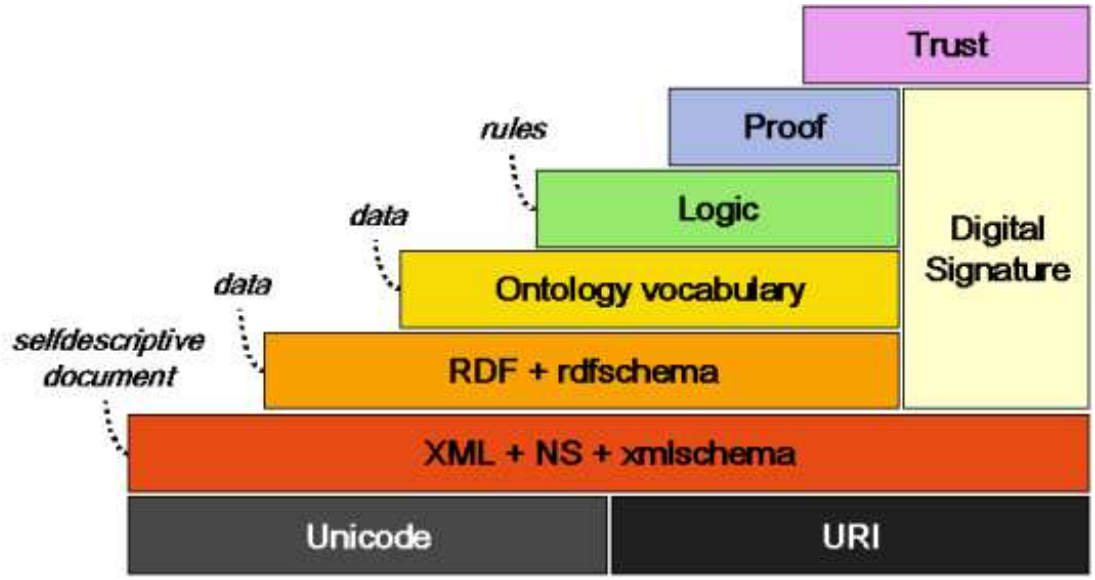

Fig.2.2. The layers of the Semantic Web [Berners-Lee]

Unicode codes the characters of all the major languages in use today.[UNICODE].

Layer 2. **XML**, namespaces and **XML** Schema

See 2.2. for a description of **XML**.

Layer 3. **RDF** en **RDF** Schema

The first two layers consist of basic internet technologies. With layer 3 the Semantic Web begins.
RDF Schema (rdfs) has as a purpose the introduction of some basic ontological notions. An example is the definition of the notion “Class” and “subClassOf”.

Layer 4. The ontology layer

The definitions of rdfs are not sufficient. A more extensive ontological vocabulary is needed. This is the task of the Web Ontology workgroup of the **W3C** who has defined already **OWL** (Ontology Web Language) and **OWL** Lite (a subset of **OWL**).

Layer 5. The logic layer

In the case study the use of rules was mentioned. For expressing rules a logic layer is needed. An experimental logic layer exists [SWAP,CWM]. This layer is treated in depth in the following chapters.

Layer 6. The proof layer

In the vision of Tim Berners-Lee the production of proofs is not part of the Semantic Web. The reason is that the production of proofs is still a very active area of research and it is by no means possible to make a standardisation of this. A Semantic Web engine should only need to verify proofs. Someone sends to site A a proof that he is authorised to use the site. Then site A must be able to verify that proof. This is done by a suitable inference engine. Three inference engines that use the rules that can be defined with this layer are: **CWM** [SWAP/CWM], Euler [DEROO] and **RDFE**ngine developed as part of this thesis.

Layer 7. The trust layer

Without trust the Semantic Web is unthinkable. If company B sends information to company A but there is no way that A can be sure that this information really comes from B or that B can be trusted then there remains nothing else to do but throw away that information. The same is valid for exchange between citizens. The trust has to be provided by a web of trust that is based on cryptographic principles. The cryptography is necessary so that everybody can be sure that his communication partners are who they claim to be and what they send really originates from them. This explains the column “Digital Signature” in fig. 2.2. The trust policy is laid down in a “facts and rules” database. This database is used by an inference engine like **RDFE**ngine. A user defines his policy using a **GUI** that produces a policy database. A policy rule might be e.g. *if* the virus checker says ‘**OK**’ *and* the format is .exe *and* it is signed by ‘TrustWorthy’ *then* accept this input.
The impression might be created by fig. 2.2 that this whole layered building has as purpose to implement trust on the internet. Indeed it is necessary for implementing trust but, once the pyramid of fig. 2.2 comes into existence, on top of it all kind of applications can be built.

Layer 8. The applications

This layer is not in the figure; it is the application layer that makes use of the technologies of the underlying 7 layers. An example might be two companies A and B exchanging information where A is placing an order with B.

3. Related work

3.1. Automated reasoning

3.1.1. Introduction

After an overview of systems of *automated reasoning*, the strategy followed in this thesis is briefly explained.
In connection with automated reasoning the terms *machinal reasoning* and, in a more restricted sense, *theorem provers*, are also relevant.
There are proof checkers i.e. programs that control the validity of a proof and proof generators.In the semantic web an inference engine will not necessarily serve to generate proofs but to check proofs; those proofs must be in a format that can be easily transported over the World Wide Web.

3.1.2. General remarks

Automated reasoning is an important domain of computer science. The number of applications is constantly growing.

- proof of theorems in mathematics
- reasoning by intelligent agents
- natural language understanding
- mechanical verification of programs
- hardware verifications (also chips design)
- planning
- a proof checker controls a proof made by another system
- and ... whatever problem that can be logically specified (a lot!)

Generally, in automated reasoning there is a *database of expressions and a logic* system. Whenever a *lemma* has to be proved, the logic system tries to use the expressions in order to deduce the lemma.
In the Semantic Web the lemma is called a *query*. An *answer* is either the confirmation of the query if the query does not contain variables; or it is the query with the variables substituted with terms.
Hilbert-style calculi have been traditionally used to characterize logic systems. These calculi usually consist of a few axiom schemata and a small number of rules that typically include modus ponens and the rule of substitution.
[STANFORD]

3.1.3. Reasoning using resolution techniques

3.1.3.1. Substitutions

When a substitution *(v, t)* where *v* is a variable and *t* is a term is applied to an expression, all occurrences in the expression of the variable *v* will be replaced by the term *t*.
Example: the substitution *(x," son")* applied to *("John", x, "Paul")* will give *("John", "son", "Paul")*.

The application of a substitution *s* to a term *t* is written *st*.
When s_1 and s_2 are substitutions and *t* is a term, then $s_1 s_2 t$ is a subtitution too.

3.1.3.2. Unification

The substitution *s* unifies expressions e_1 and e_2 if $s\ e_1 = s\ e_2$. It is possible that two expressions are unified by more than one substitution. The *most general unifier* of two expressions is the substitution *s* such that, if s_1 is also a unifier of these expressions, $s_1 = ss_x$.

3.1.3.3. Resolution reasoning

3.1.3.3.1. Introduction

In resolution reasoning the logic system tries to proof the negation of the query. If this proof has as a result the value *false* then the query must be *true*. This is the *refutation* of the negation of the query. For a certain logic system it must be proven that the system is *refutation complete* i.e. that it is always possible to proof that an expression is false.

3.1.3.3.2. The resolution principle

Resolution reasoning uses the resolution principle.

First two simple examples:

1) $a \rightarrow b$ can be written $\neg a \vee b$.
Given $a \rightarrow b$, *a* and the query *b*. The query is negated to $\neg b$, what must be proved to be false.
From $\neg a \vee b$ and $\neg b$ follows $\neg a$. But from $\neg a$ and *a* follows *false* what had to be proved.

2) from $\neg Af(x) \vee \neg Rh(x)y \vee Ay$ and $Rzf(u) \vee Bz$ follows
$\neg Af(x) \vee Af(u) \vee Bh(x)$ with unifying substitution
$s = [(z, h(x)),(y, f(u))]$.
After subtitution the factors $\neg Rh(x)f(u)$ and $Rh(x)f(u)$ cancel each other.

Resolution principle

If *A* and *B* are two expressions and *s* is a substitution unifying two factors *L1* and *L2* and *L1* occurs in *A* and $\neg L2$ occurs in *B*, then the resolution principle gives $s((A \setminus L1) \vee (B \setminus L2))$ where the anti-slash stands for "without".
[VANBENTHEM]

3.1.3.3.2. The resolution strategy

The mechanism by which the refutation of the negated query is obtained consists of repeatedly applying the resolution principle [STANFORD]. The resolution principle has to be applied to two expressions. The choice of these expressions is part of the *resolution strategy.* Another example of mechanisms that are part of the resolution strategy are the removal of redundant expressions or tautologies. *Subsumption* is the replacement of specific expressions by more general ones. Whatever the resolution strategy, its refutation completeness must be proved.

A well known resolution strategy is *SLD-resolution*: Selection, Linear, Definite [CS]. **SLD**-resolution is used by *Prolog.* To explain **SLD**-resolution first some new notions have to be introduced.
A *term* is:
a) an individual variable or a constant
b) if *f* indicates a function and *t1, …,tn* are terms, then *f(t1, …,tk)* is a term
c) nothing else is a term
[VANBENTHEM].
An *atom* is a formula of the form *P(t1, t2, … tn)* where *t1, .., tn* are terms. A *literal* is an atom (*positive literal)* or the negation of an atom (*negative literal)*. With literals *clauses* are formed. A clause is a disjunction of literals: $L1 \vee L2 \ldots \vee Lm$ where *L1, …, Lm* are literals.

Example of a clause: $P(a,b) \vee Z(D(b,c),G) \vee X$.

A clause is universally quantified i.e. all variables are universal variables. A *Horn clause* has the form:
$\neg A1 \vee \neg A2 \vee \ldots \neg An \vee A$ where *Ax* are positive literals and all variables are universally quantified.
[VANBENTHEM]

Now back to **SLD**.
In **SLD** a *selection function* exists for clauses; a quite simple one given the fact that the first in the list is selected. This is one of the points where optimization is possible namely by using another selection function.
Linear is explained here below.
Prolog works with Horn clauses.
In Prolog the definition of *a definite database* is a database that has exactly one positive literal in each clause and the unification is done with this literal.
Following resolution strategies are respected by an **SLD**-engine [CS]:

- *depth first*: each alternative is investigated until a unification failure occurs or until a solution is found. The alternative to depth first is breadth first.
- *set of support*: at least one parent clause must be from the negation of the query or one of the "descendents" of a query clause. This is a complete procedure that gives a goal directed character to the search.
- *unit resolution*: at least one parent clause must be a "unit clause" i.e. a clause containing a single literal.
- *input resolution*: at least one parent comes from the set of original clauses (from the axioms and the negation of the goals). This is not complete in general but complete for Horn clause databases.
- *linear resolution*: one of the parents is selected with set of support strategy and the other parent is selected by the input strategy.
- *ordered resolution*: this is the way prolog operates; the clauses are treated from the first to the last and each single clause is unified from left to right.

3.1.3.4. Comparison with RDFEngine

The mechanism used in **RDFE**ngine is resolution based reasoning with a **SLD**-resolution strategy. The proof of the refutation completeness (chapter 5) is given by reasoning on a graph instead of using first order logic.

3.1.4. Backward and forward reasoning

Suppose a database consisting of expressions *A, B, C …* that do not contain variables and rules *(X1,…,Xn)* → *Y* where *X1, ..Xn, Y* are expressions that can contain variables. *X1, …,Xn* are called the *antecedents, Y* is the *consequent*. Let Q be a query with or without variables.
In *forward reasoning* a check is done whether the query *Q* can be unified with a fact. If this is the case then a solution is found. Then the rules are used to generate new facts *Yi, …Yn* by those rules whose antecedents can be unified with facts. After all rules have been used, again a check is done for unifying the query with a fact. Again solutions can be found. Then the rules are used again

etc… This process goes on till no more new facts can be deduced by the rules. All solutions will then have been found.

In *backward reasoning* a check is also done whether *Q* can be unified with a fact. After that however the system tries to unify *Q* with the consequent of a rule. Suppose *Q* unifies with the consequent *Y* of the rule *(X1,…,Xn)* → *Y* with the susbtitution *s*. Then *sY* will be true if *sX1, …,sXn* are true. Important here is that the rule is interpreted *backward*. Resolution or backward reasoning applied to **RDF** will be explained in detail in chapter 5.

3.1.5. Other mechanisms

There are many other mechanisms for automated reasoning. I will only list some of them with references.

- sequent deduction [STANFORD] [VAN BENTHEM].
- natural deduction [STANFORD] [VAN BENTHEM].
- the matrix connection method [STANFORD].
- term rewriting [DICK].
- mathematical induction [STANFORD] [WALSH].
- higher order logic [STANFORD].
- non-classical logics [STANFORD].
- lambda calculus [GUPTA] [HARRISON].
- proof planning [BUNDY].

3.1.6. Theorem provers

Three different kinds of provers can be discerned:

- those that want to mimic the human thought processes
- those that do not care about human thought, but try to make optimum use of the machine
- those that require human interaction.

There are domain specific and general theorem provers. Provers might be specialised in one mechanism e.g. resolution or they might use a variety of mechanisms.

In general theorem provers are used to solve difficult or complex problems in the realm of exact sciences and often those are problems that are difficult to handle manually.

[NOGIN] gives the following ordering:

- higher-order interactive provers:
    - constructive: **ALF**, Alfa, Coq, (Meta**PRL**, Nu**PRL**)
    - classical: **HOL**, **PVS**
- logical Frameworks: Isabelle, **LF**, Twelf, (Meta**PRL**)
- inductive provers: **ACL**2, Inka
- automated:
    - multi-logic: Gandalf, **TPS**
    - first-order classical logic: Otter, Setheo, **SPASS**
    - equational reasoning: **EQP**, Maude
- other: Omega, Mizar

3.1.7. Conclusion

There are a lot of possibilities for making a choice of a basic mechanism for inferencing for the Semantic Web. There is a substantial difference between the goals of a theorem prover and a basic engine for the Semantic Web. Such an engine should have following characteristics:

- usable for a broad spectrum of applications
- able to handle large data sets what implies a simple structure
- usable for subquerying what implies an inference process that can be interrupted

In first instance **RDF** with the addition of an implication could be sufficient. As inference mechanism an **SLD**-engine seems most appropriate for following reasons:

- it has already been used intensively for a broad spectrum of applications and specifically by Prolog programs
- it has been used for access to databases [ALVES].
- the refutation process and the corresponding goal directed reasoning (starting with the query) is most suitable for an interrupted inference process

3.2. Logic

3.2.1. First order logic

An obvious way to define inferencing on top of **RDF** is to make a description of **RDF** in first order logic. Then rules are defined by an implication (see e.g. [DECKER]). Reasoning is done using first order logic. In **RDFE**ngine a constructive approach is used instead of an approach by first order logic.

3.2.2. Intuitionistic or constructive logic

In intuitionistic logic the meaning of a statement resides not in its truth conditions but in the means of proof or verification. In classical logic *p v ~p* is always true ; in constructive logic *p v ~p* has to be 'constructed'. If *forSome x.F* then effectively it must be possible to compute a value for *x*.
The **BHK**-interpretation of constructive logic (Brouwer, Heyting, Kolmogorov) [STANFORD, ARTEMOV1, ARTEMOV2]:

a) a proof of *A* and *B* is given by presenting a proof of *A* and a proof of *B*.
b) a proof of *A* or *B* is given by presenting either a proof of *A* or a proof of *B*.
c) a proof of *A* ➔ *B* is a procedure which permits us to transform a proof of *A* into a proof of *B*.
d) the constant false has no proof.

A proof of ~*A* is a procedure that transforms a hypothetical proof of *A* into a proof of a contradiction.
For two reasons *verifiability* is important for the Semantic Web.
a) the processes are fully automated (no human intervention) so the verification has to be done by computers.
b) different parts of the inferencing proces (inference steps) are executed by different engines on different sites.
RDFEngine has a constructive view on **RDF** in that every triple in an inference process is constructed and is thus a verifiable triple (2.4.2). I argue that this constructive view is necessary in automated processes.

3.2.3. Paraconsistent logic

3.2.3.1. Elaboration

The development of *paraconsistent logic* was initiated in order to challenge the logical principle that anything follows from contradictory premises, *ex contradictione quodlibet (ECQ)* [STANFORD]. Let |= be a relation of logical consequence, defined either semantically or proof-theoretically. Let us say that |= is *explosive* iff for every formula *A* and *B*, {*A* , ~*A*} |= *B*. Classical logic, intuitionistic logic, and most other standard logics are explosive. A logic is said to be *paraconsistent* iff its relation of logical consequence is not explosive.
Also in most paraconsistent systems the disjunctive syllogism does not hold: From *A, ~ A v B* we cannot conclude *B*. Some systems:
Non-adjunctive systems: from *A , B* the inference *A & B* fails.
Non-truth functional logic: the value of *A i*s independent from the value of ~*A* (both can be one e.g.).
Many-valued systems: more than two truth values are possible.If one takes the truth values to be the real numbers between 0 and 1, with a suitable set of designated values, the logic will be a natural paraconsistent fuzzy logic.

3.2.3.2. Relevance for the Semantic Web

It is to be expected that contradictions will not be exceptional in the Semantic Web. It would not be acceptable under such circumstances that anything can be concluded from a contradiction. Therefore the logic of the Semantic Web must not be explosive.

3.2.4. Conclusion

Constructive logic and paraconsistent logic are important for the Semantic Web. Many other logics have the potential to be used in the Semantic Web. Some candidates are: first order logic, higher order logics, modal logic, linear logic.

3.3. Existing software systems

3.3.1. Introduction

I make a difference between a *query engine* that just does querying on a **RDF** graph but does not handle rules and an *inference engine* that also handles rules. In the literature this difference is not always so clear.
The complexity of an inference engine is a lot higher than a query engine. The reason is that rules permit to make sequential deductions. In the execution of a query this deductions are to be constructed. This is not necessary in the case of a query engine.
I will not discuss query engines. Some examples are: **DQL** [DQL], **RQL** [RQL], **XQUERY** [BOTHNER].
Rules also suppose a logic base that is inherently more complex than the logic in the situation without rules. For an **RDF** query engine only the simple principles of entailment on graphs are necessary (see chapter 5).
RuleML is an important effort to define rules that are usable for the World Wide Web [GROSOP].
The Inference Web [MCGUINESS2003] is a recent realisation that defines a system for handling different inferencing engines on the Semantic Web.

3.3.2. Inference engines

3.3.2.1.Euler

Euler is the program made by De Roo [DEROO]. It does inferencing and also implements a great deal of **OWL** (Ontology Web Language).
The program reads one or more triple databases that are merged together and it also reads a query file. The merged databases are transformed into a linked structure (Java objects that point to other Java objects). The philosopy of Euler is graph oriented.

This structure is different from the structure of **RDFE**ngine. In **RDFE**ngine the graph is not described by a linked structure but by a collection of triples, a tripleset.
The internal mechanism of Euler is in accordance with the principles exposed in chapter 5.
De Roo also made a collection of test cases upon which I based myself for testing **RDFE**ngine.

3.3.2.2. CWM

CWM is a program made by Berners-Lee [CWM].It is based on forward reasoning. **CWM** makes a difference between existential and universal variables[BERNERS]. **RDFE**ngine does not make that difference. It follows the syntaxis of Notation 3: *log:forSome* for existential variables, *log:forAll* for universal variables. As will be explained in chapter 5 all variables in **RDFE**ngine are quantified in the sense: *forAllThoseKnown*. Blank (anonymous) nodes are instantiated with a unique **URI**.
CWM makes a difference between *?a* for a universal, local variable and *_:a* for an existential, global variable. **RDFE**ngine maintains the difference between local and global but not the quantification.

3.3.2.3. TRIPLE and RuleML

Introduction

TRIPLE and Rule**ML** are two examples of inference engines that have chosen to use an approach based on classic logic i.e. a logic that is first order logic or a subset.

TRIPLE

TRIPLE is based on Horn logic and borrows many features from F-Logic [SINTEK].
TRIPLE is the successor of Si**LRI**.

RuleML

Rule**ML** is an effort to define a specification of rules for use in the World Wide Web.
The kernel of Rule**ML** are *datalog* logic programs (see chapter 2) [GROSOF]. It is a declarative logic programming language with model-theoretic semantics. The logic is based on Horn logic.
It is a *webized* language: namespaces or defined as well as **URI**'s.
Inference engines are available.

There is a cooperation between the Rule**ML** Iniative and the Java Rule Engines Effort.

3.3.3. The Inference Web

3.3.3.1. Introduction

The Inference Web was introduced by a series of recent articles [MCGUINESS2003].
When the Semantic Web evolves it is to be expected that a variety of inference engines will be used on the web. A software system is needed to ensure the compatibility between these engines.
The inference web is a software system consisting of:
a web based registry containing details on information sources and reasoners called the Inference Web Registry.

1) an interface for entering information in the registry called the Inference Web Registrar.
2) a portable proof specification
3) an explanation browser.

3.3.3.2. Details

In the Inference Web Registry data about inference engines are stored. These data contain details about *authoritative sources, ontologies, inference engines and inference rules.* In the explanation of a proof every inference step should have a link to at least one inference engine.

1) the Web Registrar is an interface for entering information into the Inference Web Registry.
2) the portable proof specification is written in the language **DAML+OIL**. In the future it will be possible to use **OWL**. There are four major components of a portable proof:
    a) inference rules
    b) inference steps
    c) well formed formulae
    d) referenced ontologies

These are the components of the inference process and thus produces the proof of the conlusion reached by the inferencing.

3) the explanation browser shows a proof and permits to focus on details, asks additional information, etc…

3.3.3.3. Conclusion

I believe the Inference Web is a major and important step in the development of the Semantic Web. It has the potential of allowing a cooperation between different inference engines. It can play an important part in establishing trust by giving explanations about results obtained with inferencing.

3.3.4. Swish

Swish is a software package that is intended to provide a framework for building programs in Haskell that perform inference on **RDF** data [KLYNE]. Where **RDFE**ngine uses resolution backtracking for comparing graphs, Swish uses a graph matching algorithm described in [CARROLL]. This algorithm is interesting because it represents an alternative general way, besides forwards and backwards reasoning, for building a **RDF** inference engine.

Chapter 4. RDF and logic

4.1. Introduction

In this chapter the logic properties of **RDF** will be explored. In the first place the properties of **RDF** are compared with first order logic. Then a comparison is made with Horn clauses, where it is shown that most Horn clauses can be translated to **RDF** triples.
It will be demonstrated that **RDF** with an added implication can be given an interpretation in constructive logic. It is then necessary however to give the implication a constructive definition.

4.2. Differences between RDF and FOL

4.2.1. Anonymous entities

Take the following declarations in first order logic (**FOL**):

$\exists car: owns(John, car) \wedge brand(car, Ford)$
$\exists car:brand(car, Opel)$

and the following **RDF** declarations:

(John, owns, car)
(car, brand, Ford)
(car, brand, Opel)

In **RDF** this really means that John owns a car that has two brands (see also the previous chapter). This is the consequence of the graph model of **RDF**. A declaration as in **FOL** where the atom car is used for two brands is not possible in **RDF**. In **RDF** things have to be said a lot more explicitly:

(John, owns, Johns_car)
(Johns_car, is_a, car)
(car, has, car_brand)
(Ford, is_a, car_brand)
(Opel, is_a, car_brand)
(car_brand, is_a, brand)

This means that it is not possible to work with anonymous entities in **RDF**.
On the other hand given the declaration:

$\exists car:color(car, yellow)$

it is not possible to say whether this is Johns car or not.
In **RDF**:

color(Johns_car, yellow)

So here it is necessary to add to the **FOL** declaration:

$\exists$ *car:color(car, yellow)* $\wedge$ *owns(John, car).*

4.2.2.'not' and 'or'

Negation and *disjunction* are two notions that have a connection. If the disjunction is *exclusive* then the statement: *a cube is red or yellow* implies that, if the cube is red, it is *not*_yellow, if the cube is yellow, it is *not*_red.
If the disjunction is *not exclusive* then the statement: *a cube is red or yellow* implies that, if the cube is red, it might be *not*_yellow, if the cube is yellow, it might be *not*_red.

Negation by failure means that, if a fact cannot be proved, then the fact is assumed not to be true [ALVES]. In a *closed world assumption* this negation by failure is equal to the logical negation. In the internet an *open world assumption* is the rule and negation by failure should be simply translated as: *not found.*
It is possible to define a not as a property. $\neg$ *owner(John, Rolls)* is implemented as: *not_owner(John, Rolls)*.
I call this *negation by declaration*. Contrary to negation by failure negation by declaration is monotone and corresponds to a **FOL** declaration.
Suppose the **FOL** declarations:

$\neg$*owner(car, X)* $\rightarrow$ *poor(X).*
owner(car, X) $\rightarrow$ *rich(X).*
$\neg$*owner(car, Guido).*

and the query:

poor(Guido).

In **RDFP**rolog this becomes:

not_owner(car, X) :> poor(X).
owner(car, X) :> rich(X).
not_owner(car, Guido).

and the query:

poor(Guido).

The negation is here really tightly bound to the property. The same notation as for **FOL** could be used in **RDFP**rolog with the condition that the negation is bound to the property.
Now lets consider the following problem:

poor(X) :> not_owner(car, X).
rich(X) :> owner(car, X).
poor(Guido).

and the query:

owner(car, Guido).

RDFEngine does not give an answer to this query. This is the same as the answer 'don't know'. A forward reasoning engine would find: *not_owner(car, Guido),* but this too is not an answer. However when the query is: *not_owner(car, Guido)* the answer is 'yes'. So the response really is: *not_owner(car, Guido).* So, when a query fails, the engine should then try the negation of the query. When that fails also, the answer is really: 'don't know'.
This implies a tri-valued logic: a query is true i.e. a solution can be constructed, a query is false i.e. a solution to the negation of the query can be constructed or the answer to a query is just: I don't know.

The same procedure can be followed with an *or*. I call this *or by declaration*.
Take the following **FOL** declaration: *color(yellow) ∨ color(blue).*
With 'or by declaration' this becomes:

color(yellow_or_blue).
color(yellow), color(not_blue) → color(yellow_or_blue).
color(not_yellow), color(blue) → color(yellow_or_blue).
color(yellow), color(blue) → color(yellow_or_blue).

Note that a 'declarative not' must be used and that three rules have to be added.
An example of an implementation of these features can be found at 5.17.4., the example of the Alpine Sports Club.
This way of implementing a disjunction can be generalized. As an example take an Alpine club. Every member of the club is or a skier, or a climber. This is put as follows in **RDFP**rolog:

property(skier).
property(climber).
class(member).
subPropertyOf(sports, skier).
subPropertyOf(sports, climber).
type(sports, or_property).
climber(John).
member(X) :> sports(X).
subPropertyOf(P1, P), type(P1, or_property), P(X) :> P1(X).

The first rule says: if *X* is a member then *X* has the property sports.
The second rule says: if a property *P1* is an or_property and *P* is a subPropertyOf *P1* and *X* has the property *P* then *X* also has the property *P1*.
The query: *sports(John)* will be answered positively.
The implementation of the disjunction is based on the creation of a higher category which is really the same as is done in **FOL**:
$\forall x$*:skier(x)* $\vee$ *climber(x)*,
only in **FOL** the higher category remains anonymous.
A conjunction can be implemented in the same way. This is not necessary as conjunction is implicit in **RDF**.
The disjunction could also be implemented using a builtin e.g.:
member(club, X) :> or(X, list(skier, climber)).
Here the predicate or has to be interpreted by the inference engine, that needs also a buitlin list predicate. The engine should then verify whether *X* is a skier or a climber or both.

4.2.3. Proposition

I propose to extend **RDF** with a negation and a disjunction in a constructive way:
Syntactically the negation is represented by a not predicate in front of a predicate in **RDFP**rolog:
not(a(b, c)).
and in Notation 3 by a not in front of a triple:
not{:a :b :c.}.
Semantically this negation is tied to the property and is identical to the creation of an extra property *not_b* that has the meaning of the negation of *b*. Such a triple is only true when it exists or can be deduced by rules.
Syntactically the disjunction is implemented with a special ‘builtin’ predicate *or*.
In **RDFP**rolog if the or is applied to the subject:
p(or(a, b), c).
or to the object:
p(a, or(b, c)).
or to the predicate:

or(p, p1)(a, b).
and in Notation 3:
{:a :p or(:b, :c)}.
In **RDFP**rolog this can be used in place of the subject or object of a triple; in Notation 3 in place of subject, property or object.
Semantically in **RDFP**rolog the construction *p(a, or(b, c))* is true if there exists a triple *p(a, b)* or there exists a triple *p(a, c)*.
Semantically in Notation 3 the construction *{:a :p or(:b, :c)}.* is true when there exists a triple *{:a :p :b}* or there exists a triple *{:a :p :c}*.
It is my opinion that these extensions will greatly simplify the translation of logical problems to **RDF** while absolutely not modifying the semantic model of **RDF**.
I also propose to treat negation and disjunction as properties. For negation this is easy to understand: a thing that is not yellow has the property not_yellow. For disjunction this can be understood as follows: a cube is yellow or black. The set of cubes however has the property yellow_or_black. In any case a property is something which has to be declared. So a negation and disjunction only exist when they are declared.

4.3. Logical implication

4.3.1. Introduction

Implication might seem basic and simple; in fact there is a lot to say about. Different interpretations of implication are possible. A constructive interpretation is chosen for **RDFE**ngine.

4.3.2. Problem

In 3.2. I argued that triples in a proof should be verifiable. So when an implication is added to **RDF** as an extension, the triples that can be deduced by the implication should only be verifiable triples.

4.3.3. RDF and implication

Material implication is implication in the classic sense of first order logic. In this semantic interpretation of implication the statement ‘If dogs are reptiles, then the moon is spherical’ has the value ‘true’. There has been a lot of criticism about such interpretation. The critic concerns the fact that the premise has no connection whatsoever with the consequent. This is not all. In the statement ‘If the world is round then computer scientists are good people’ both the antecedent and the consequent are, of course, known to be true, so the statement is true. But there is no connection whatsoever between premise and consequent.

In **RDF** I can write:
(world, a, round_thing) implies (computer_scientists, a, good_people).
This is good, but I have to write it in my own namespace, let's say: *namespace = http://guido.naudts.* Other people on the World Wide Web might judge:
Do not trust what this guy Naudts puts on his web site.
So the statement is true but…
Anyhow, antecedents and consequents do have to be valid triples i.e. the **URI**'s constituting the subject, property and object of the triples do have to exist. There must be a real and existing namespace on the World Wide Web where those triples can be found. If e.g. the consequent generates a non-existing namespace then it is invalid. It is possible to say that invalid is equal to *false,* but an invalid triple just will be ignored. An invalid statement in **FOL** i.e. a statement that is *false* will not be ignored.

Strict implication or *entailment* : if $p \rightarrow q$ then, necessarily, if p is true, q must be true. Given the rule *(world, a, round_thing) implies (computer_scientists, a, good_people)*, if the triple *(world, a, round_thing)* exists then, during the closure process the triple *(computer_scientists, a, good_people)* will actually be added to the closure graph. So, the triple exists then, but is it true? This depends on the person or system who looks at it and his trust system.

Material implication might produce non verifiable triples. What is wanted is that, if the antecedents of the rule can be unified with 'true' triples, then the substitution applied to the consequents will give necessarily 'true' triples. This is then strict implication.

In **RDF** with implication the *modus ponens* does not entail the corresponding *modus tollens*. In classic notation: $a \rightarrow b$ does not entail $\neg b \rightarrow \neg a$. If a constructive negation as proposed before is accepted, then this deduction of modus tollens from modus ponens could be accepted.

4.3.3. Conclusion

The implication to be defined as extension for **RDF** should be a *strict implication.*

4.4. RDF and constructive logic

RDF is more 'constructive' than **FOL**. Resources have to exist and receive a name. Though a resource can be anonymous in theory, in practical inferencing it is necessary to give it an anonymous name.
A negation cannot just be declared; it needs to receive a name. Saying that something is not yellow amounts to saying that it has the property not_yellow.

The set of all things that have either the property yellow or blue or both needs to be declared in a superset that has a name.
For a proof of $A \vee \neg A$ either A has to be proved or $\neg A$ has to be proved (e.g. an object is yellow or not_yellow only if this has been declared).
This is the consequence of the fact that in the **RDF** graph everything is designated by a **URI**.
A proof of a triple is given when:
a) it is a valid triple
b) it is accepted by the trust system
The **BHK**-interpretation of constructive logic states:
(Brouwer, Heyting, Kolmogorov) [STANDARD]

a) "A proof of *A* and *B* is given by presenting a proof of *A* and a proof of *B*."

   With **RDF** all triples in a set of triples have between them an implicit conjunction. If *A* and *B* are triples in the **RDF** database, they are necessarily true and their conjucntion is thus constructively true too.

b) "A proof of *A* or *B* is given by presenting either a proof of *A* or a proof of *B* or both."

   There is no disjunction defined in **RDF**. Would it be defined as above(3.2), then it would be constructive .

c) "A proof of $A \rightarrow B$ is a procedure which permits us to transform a proof of *A* into a proof of *B*."

   The implication is an extension of **RDF**. I propose a constructive implication where $(A1, A2, \ldots, An) \rightarrow B$ is interpreted: if (*A1, A2, …An)* are true triples then *B* is a true triple.

d) "The constant false has no proof."

   The constant false does not exist in **RDF** but it could be added .

This constructive approach is chosen for reasons of verifiability. Suppose an answer to a query is $T1 \vee T2$ where *T1* and *T2* are triples but neither *T1* or *T2* can be proven. How can such an answer be verified?
When solutions are composed of existing triples then these triples can be verified i.e. the originating namespace can be queried for a confirmation or perhaps the triples or their constituents (subject, predicate, object) could be signed.

4.5. The Semantic Web and logic

4.5.1. Discussion

One of the main questions to consider when speaking about logic and the Semantic Web is the logical difference between an *open world* and *a closed world*.

In a closed world it is assumed that everything is known. A query engine for a database e.g. can assume that all knowledge is in the database. Prolog also makes this assumption. When a query cannot be answered Prolog responds: "No", where it could also say: "I don't know".
In the world of the internet it is not generally possible to assume that all facts are known. Many data collections or web pages are made with the assumption that the user knows that the data are not complete.
These collections of data are considered to be open. An example is the set of all large internet sites e.g. sites with more than 100.000 **HTML** pages. At any moment there is a finite number of such sites but this number is never known.There are some fundamental implications of this fact:

1) it is impossible to take the complement of a collection as: a) the limits of the collection are not known and b) the limits of the universe of discourse are not known. Example: the number of large internet sites is not known and the number of sites in the complement set is not known either.
2) negation can only be used when it is explicitly declared. Example: it is not possible to speak of all sites that are not large, because this set is not known.It is not possible to speak of all resources that are not yellow. It is possible to say that resource *x* is not yellow or that all elements of set *y* are not yellow.
3) the universal quantifier does not exist. It is impossible to say : for all elements in a set or in the universe as the limits of those entities are not known. The existential quantifier, *for some,* really means : *for all those that are known.*
4) the disjunction must be constructive. A proof of *a or b* is a proof of *a*, a proof of *b* or a proof of *a* and a proof of *b*. In a closed world it can always be determined whether *a or b* is true.
   Example: 'a site is in the set of large sites or it is not'. In general the truth value of this expression cannot be determined; it can only be determined if the size of the site is known.
5) four sets can be discerned: the open set, the complement of the open set, the set containing the elements of which the membership is not known and the set of unknown elements. The open set and its complement contain only elements of which the membership is known.

Lets again review the 5 points above but for a closed set. An example is the list of all members of a club.

1) for a closed set it is possible to say from any item: it is in the set or it is not. But the complement of the set is an open set.
   Example: a person is member of the club or not. But all persons that are not member of the club are not known.
2) negation can be used to define elements of the complement e.g. all persons that are not members of the club must pay for entrance.
3) the universal quantifier exists e.g. for all members of the club, give a reduction.
4) the disjunction is as in **FOL**. If *a* or *b* are known then the truth value of $a \vee b$ is also known. This is the law of the excluded middle.
   Example: a person is member or he is not. For any person this expression is always true.
5) there are only three sets involved. Example: the set of the members of the club, the set of those who are not a member and the set of the unknown persons.

4.5.2. Consequences

The consequence of the above is that an inference engine for the Semantic Web must handle closed sets and open sets in a different way because the logic is different. A default could be that sets are supposed to be open. A set is considered closed when this is explicitly declared.
RDFEngine considers sets to be open.

4.6. Monotonicity

Another aspect of logics that is very important for the Semantic Web is the question whether the logic should be *monotonic* or *nonmonotonic.*
Under monotonicity in regard to **RDF** I understand that, whenever a query relative to an **RDF** file has a definite answer and another **RDF** file is merged with the first one , the answer to the original query will still be obtained after the merge. As shown before there might be some extra answers and some of those might be contradictory with the first answer. However this contradiction will clearly show.
If the system is nonmonotonic some answers might dissappear; others that are contradictory with the first might appear. All this would be very confusing.

Here follows an example problem where, apparently, it is impossible to get a solution and maintain the monotonicity.
I state the problem with an example.
A series of triples:

(item1, price, price1), (item2, price, price2), etc…

is given and a query is done asking for the lowest price.
Then some more triples are added:

(item6, price, price6), (item7, price, price7), etc…

and again the query is done asking for the lowest price.
Of course, the answer can now be different from the first answer. Monotonicity would require the first answer to remain in the set of solutions. This obviously is not the case.
Clearly however, a Semantic Web that is not able to perform an operation like the above, cannot be sactisfactory.

I will give an encoding of the example above in **RDFP**rolog and then discuss this further.

List are encoded with *rdf:first* for indicating the head of a list and *rdf:rest* for indicating the rest of the list. The end of a list is encoded with *rdf:nil*.

I suppose the prices have already been entered in a list what is in any case necessary.

rdf:first(l1, ”15”).
rdf:rest(l1, l2).
rdf:first(l2, ”10”).
rdf:rest(l2, rdf:nil).

rdf:first(L, X), rdf:rest(L, rdf:nil) :> lowest(L, X).
rdf:first(L, X), rdf:rest(L, L1), lowest(L1, X1), lesser(X, X1) :> lowest(L, X).
rdf:first(L, X), rdf:rest(L, L1), lowest(L1, X1), lesser(X1, X) :> lowest(L, X1).

The query is:
lowest(l1, X).

giving as a result:
lowest(l1, ”10”).

Let’s add two prices to the beginning of the list. The list then becomes:
rdf:first(l3, ”7”).
rdf:rest(l3, l4).
rdf:first(l4, ”23”).
rdf:rest(l4, l1).
rdf:first(l1, ”15”).

rdf:rest(l1, l2).
rdf:first(l2, ”10”).
rdf:rest(l2, rdf:nil).

The query will now give as a result:
lowest(l1, ”7”).

Monotonicity is not respected. The reason for this is, that the predicate *lesser* is a mathematical builtin that has to be interpreted by the inference engine and that breaks the monotonicity.
Some operations apparently cannot be executed while at the same time respecting monotonicity.
Clearly, however, these operations might be necessary. Whenever a builtin is added however, the characteristics of the logic system can be changed.

4.7. From Horn clauses to RDF

4.7.1. Introduction

Doubts have often been uttered concerning the expressivity of **RDF**. An inference engine for the World Wide Web should have a certain degree of expressiveness because it can be expected, as shown in chapter one, that the degree of complexity of the applications could be high.
I will show in the next section that all Horn clauses can be transformed to **RDF**, perhaps with the exception of functions. This means that **RDF** has at least the same expressiveness as Horn clauses.

4.7.2. Elaboration

For Horn clauses I will use the Prolog syntax; for **RDF** I will use **RDFP**rolog.
I will discuss this by giving examples that can be easily generalized.

0-ary predicates:
Prolog: *venus.*
RDF: *Tx(Ty, venus).*
where *Ty* and *Tx* are created anonymous **URI**’s.
Unary predicates:
Prolog: *name(“John”).*
RDF: *name(Ty, “John).*
Binary predicates:
Prolog: *author(book1, author1).*
RDF: *author(book1, author1).*

Ternary and higher:

Prolog: *sum(a, b, c).*
RDF: *sum1(Ty, a).*
sum2(Ty, b).
sum3(Ty, c).
where *Ty* represents the sum.

Embedded predicates: I will give an extensive example.
In Prolog:

```
diff(plus(A, B), X, plus(DA, DB)) :-  diff(A, X, DA),  diff(B, X, DB).
diff(5X, X, 5).
diff(3X, X, 3).
```

and the query:

```
diff(plus(5X, 3X), X, Y).
```

The solution is: *Y = 8.*
This can be put in **RDF** but the source is rather more verbose. The result is given in fig. 4.3. Comment is preceded by #.
This is really very verbose but... this can be automated and it does not necessarily have to be less efficient because special measures are possible for enhancing the efficiency due to the simplicity of the format.
So the thing to do with embedded predicates is to externalize them i.e. to make a separate predicate.

What remains are functions:
Prolog: *p(f(x), g).*
RDF: *f(Tx, x).*
p(Tx, g).
or first: replace *p(f(x), g)* with *p(f, x, g)* and then proceed as before.
Hayes in [RDFRULES] warns however that this way of replacing a function with a relation can cause problems when the specific aspects of a function are used i.e. when only one value is needed.

The **RDF** Graph:

```
diff1(Y1, A),
diff2(Y2, X),
diff3(Y3, DA),          # diff(A, X, DA)
diff1(Y2, B),
diff2(Y2, X),
diff3(Y2, DB),          # diff(B, X, DB)
plus1(Y3, A),
plus2(Y3, B),           # plus(A, B)
diff1(Y4, Y3),
diff2(Y4, X),           #diff(plus(A, B), X,
```

```
        plus1(Y5, DA),
        plus2(Y5, DB)           # plus(DA, DB)
        :>
        diff3(Y4, Y5).          # plus(DA, DB))  -- This was the rule part.
        diff1(T1, 5X).          # here starts the data part
        diff2(T1, X).
        diff3(T1, 5).           # diff(5X, X, 5).
        diff1(T2, 3X).
        diff2(T2, X).
        diff3(T2, 3).           # diff(3X, X, 3).
        plus1(T4, 5X).
        plus2(T4, 3X).          # plus(5X, 3X).
        diff1(T5, T4).
        diff2(T5, X).           # diff(T4, X,
        plus1(T6, 5).
        plus2(T6, 3).           # plus(5, 3)

The query:
        diff3(T5, W).           # ,Y))
The answer:
        diff3(T5, T6).
```

Fig. 4.1. A differentiation in **RDF**.

4.8. Conclusion

As was shown above the logic of **RDF** is compatible with the **BHK** logic. It defines a kind of constructive logic. I argue that this logic must be the basic logic of the Semantic Web. Basic Semantic Web inference engines should use a logic that is both constructive, based on an open world assumption, and monotonic. The monotonicity can be broken, however, by buitlins.
If, in some applications, another logic is needed, this should be clearly indicated in the **RDF** datastream. In this way basic engines will ignore these datastreams.

Chapter 5. RDF, RDFEngine and inferencing

5.1. Introduction

Chapter 5 will explain the meaning of inferencing in connection with **RDF**. Two aspects will be interwoven: the theory of inferencing on **RDF** data and the specification of an inference engine, **RDFE**ngine, in Haskell.
First a specification of a **RDF** graph will be given. Then inferencing on a **RDF** graph will be explained. A definition will be given for rules, queries, solutions and proofs. After discussing substitution and unification, an inference step will be defined and an explanation will be given of the data structure that is needed to perform a step. The notion of subquerying will be explained and the proof format that is necessary for subquerying.
The explanations will be illustrated with examples and the relevant Haskell code.
RDFEngine possesses the characteristics of soundness, completeness and monotonicity. This is demonstrated by a series of lemmas.

Note: The Haskell source code presented in the chapter is somewhat simplified compared to the source code in appendix.

5.2. Recapitulation and definitions

I give a brief recapitulation of the most important points about **RDF** [RDFM,RDF Primer, RDFSC].
RDF is a system for expressing meta-information i.e. information that says something about other information e.g. a web page. **RDF** works with triples. *A triple* is composed of a *subject*, a *property* and an *object*. A *triplelist* is a list of triples. Subject, property and object can have as value a **URI**, a blank node or a triplelist. An object can also be a literal. The property is also often called *predicate*.
Triples are noted as: *(subject, property, object).* A triplelist consists of triples between '[' and ']', separated by a ','.
In the graph representation subjects and objects are *nodes* in the graph and the *arcs* represent properties. No nodes can occur twice in the graph. The consequence is that some triples will be connected with other triples; other triples will have no connections. Also the graph will have different connected subgraphs.
A node can also be a *blank* node which is a kind of anonymous node. A blank node can be *instantiated* with a **URI** or a literal and then becomes a 'normal' node. This means a blank node behaves like an existential variable with local scope. See also further in this paragraph.
A *valid RDF graph* is defined in 2.7.

Fig.5.1 gives examples of triples.

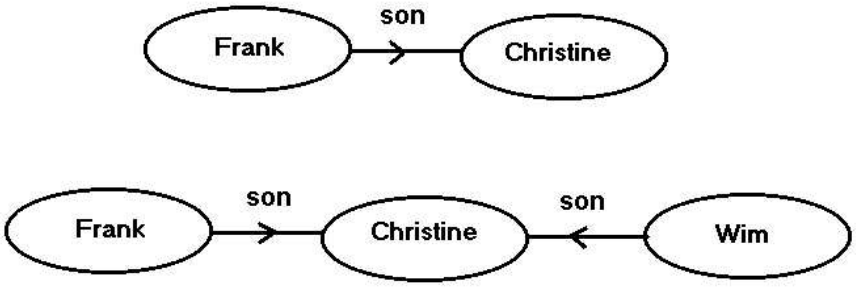

Fig.5.1. A **RDF** graph consisting of two connected subgraphs.

There are three triples in the graph of fig. 5.1: (Frank, son, Guido), (Wim, son, Christine), (Christine, daughter, Elza).

I give here the Haskell representation of triples:

data Triple = Triple (Subject, Predicate, Object) | TripleNil

type Subject = Resource
type Predicate = Resource
type Object = Resource

data Resource = URI String | Literal String | Var Vare | TripleList TripleList| ResNil

type TripleList[1] *= [Triple]*

Four kinds of variables are defined:
UVar: Universal variable with local scope
EVar: Existential variable with local scope
GVar: Universal variable with global scope
GEVar: Existential variable with global scope

-- the definition of a variable
data Vare = UVar String | EVar String | GVar String | GEVar String

These four kinds of variables are necessary as they occur in the test cases of [DEROO] that were used for testing **RDFE**ngine.

Fig.5.1 represents a **RDF** graph. Now lets make two queries. Intuitively a query is an interrogation of the **RDG** graph to see if certain facts are true or not.

[1] In most, but not all, cases the triplelist will be a tripleset. In order to simplify the text the notion list is used everywhere, even when the list represents a set.

The first query is: *[(Frank, son, Guido)]*.
The second query is: *[(?who, daughter, Elza)]*.

This needs some definitions:
A *query* is a triplelist. The triples composing the query can contain variables. When the query does not contain variables it is an **RDF** graph. When it contains variables, it is not an **RDF** graph because variables are not defined in **RDF** and are thus an extension to **RDF**. When the inferencing in **RDFE**ngine starts the query becomes the *goallist*. The goallist is the triplelist that has to be proved i.e. matched against the **RDF** graph. This process will be described in detail later.
The answer to a query consists of one or more *solutions*. In a solution the variables of the query are replaced by **URI**'s, if the query did have variables. If not, a solution is a confirmation that the triples of the query are existing triples (see also 2.7). I will give later a more precise definition of solutions.
A *variable* will be indicated with a question mark. For this chapter it is assumed that the variables are local universal variables.
A *grounded* triple is a triple of which subject and property are **URI**'s and the object is a **URI** or a literal. A grounded triplelist contains only grounded triples. A query can be grounded. In that case an affirmation is sought that the triples in the query exist, but not necessarily in the **RDF** graph: they can be deduced using rules.

Fig.5.2 gives the representation of some queries.
In the first query the question is: *is Frank the son of Guido?* The answer is of course "yes". In the second query the question is: *who is a daughter of Elza?* Here the query is a graph containing variables. This graph has to be matched with the graph in fig.5.1. So generally for executing a **RDF** query what has to be done is 'subgraph matching'.

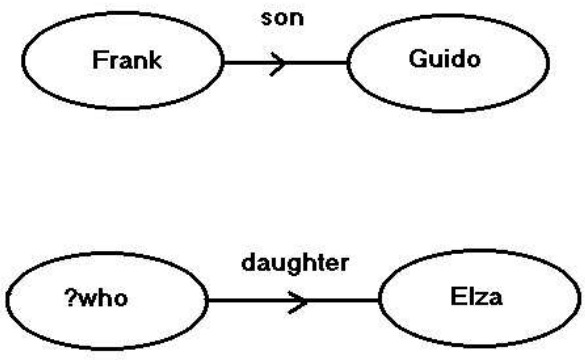

Fig.5.2. Two **RDF** queries. The first query does not contain variables.

The variables in a query are replaced in the matching process by a **URI** or a literal. This replacement is called a *substitution*. A substitution is a list of tuples

(variable, **URI**) or (variable, literal)[1]. The list can be empty. I will come back to substitutions later.

A rule is a triple. Its subject is a triplelist containing the *antecedents* and its object is a triplelist containing the *consequents*. The property or predicate of the rule is the **URI** : http://www.w3.org/2000/10/swap/log# implies. [SWAP] . I will give following notation for a rule:

{triple1, triple2,…} → {triple11, triple21,…}

For clarity, only rules with one consequent will be considered. The executable version of **RDFE**ngine can handle multiple consequents.

For a uniform handling of rules and other triples the concept *statement* is introduced in the abstract Haskell syntax:

```
type Statement = (TripleList, TripleList, String)
type Fact = Statement -- ([], Triple, String)
type Rule = Statement
```

In this terminology a rule is a statement. The first triplelist of a rule is called the *antecedents* and the second triplelist is called the *consequents*. A *fact* is a rule with empty antecedents. The string part in the statement is the *provenance* information. It indicates the origin of the statement. The origin is the namespace of the owner of the statement.
In a rule the order of the triples in the antecedents is relevant.

5.3. The graph ADT

A *RDF graph* is a set of statements. It is represented in Haskell by an *Abstract Data Type*: **RDFG**raph.hs.

The implementation of the **ADT** is done by using an *array* and a *hashtable*.

```
data RDFGraph = RDFGraph (Array Int Statement, Hash, Params)
```

The statements of the graph are in an array. The elements *Hash* and *Params* in the type *RDFGraph* constitute a *hashtable.*
A predicate is associated with each statement. This is the predicate of the first triple of the second triplelist of the statement[2]. An entry into the hashtable has as a key a predicate name and a list of numbers. Each number refers to an entry in

[1] (Theoretically also (variable, triplelist), but this is not implemented in RDFEngine)
[2] In RDFEngine all consequents have the same predicate.

the array associated with the **RDFG**raph. This structure permits to get the list of all statements that can possibly be unified with a fact i.e. have the same predicate.
Params is an array containing some parameters for the **ADT** like the last entry into the array of statements.
Example: Given the triple *(a, b ,c)* that is entry number 17 in the array, the entry in the dictionary with key 'b' will give a list of numbers […,17,…]. The other numbers will be triples or rules with the same predicate.
In this way all statements with the same associated predicate can be retrieved at once. This is used when matching triples of the query with the **RDF** graph.

The **ADT** is accessed by means of a *mini-language* defined in the module **RDFML**.hs. This mini-language contains functions for manipulating triples, statements and **RDF** Graphs. Fig. 5.3. gives an overview of the mini-language. Through the encapsulation by the mini-language this implementation of the **ADT** could be replaced by another one where only one module would need to be changed. For the user of the module only the mini-language is important[1].

[1] The ADT is written specifically for the thesis; no special effort has been done to make it generally usable.

```
Mini-language for triples, statements and graphs.

-- triple :: String -> String -> String -> Triple
-- make a triple from three strings
-- triple s p o : make a triple

nil :: Triple
-- construct a nil triple
nil = TripleNil

tnil :: Triple -> Bool
-- test if a triple is a nil triple
tnil t
      | t == TripleNil = True
      | otherwise = False

s :: Triple -> Resource
-- get the subject of a triple
s t = s1
      where Triple (s1,_,_) = t

p :: Triple -> Resource
-- get the predicate from a triple
p t = s1
      where Triple (_,s1,_) = t

o :: Triple -> Resource
-- get the object from a triple
o t = s1
      where Triple (_,_,s1) = t

st :: TripleSet -> TripleSet -> Statement
-- construct a statement from two triplesets
-- -1 indicates the default graph
st ts1 ts2 = (ts1,ts2,"-1")

gmpred :: Statement -> String
-- get the main predicate of a statement
-- This is the predicate of the first triple of the consequents
gmpred st
        | (cons st) == [] = ""
        | otherwise = pred
        where t = head(cons st)
              pred = grs(p t)

stnil :: Statement
-- return a nil statement
stnil = ([],[],"0")

tstnil :: Statement -> Bool
-- test if the statement is a nil statement
tstnil st
      | st == ([],[],"0") = True
      | otherwise = False

trule :: Statement -> Bool
-- test if the statement is a rule
trule ([],_,_) = False
trule st = True

tfact :: Statement -> Bool
```

```
-- test if the statement st is a fact
tfact ([],_,_) = True
tfact st = False

stf :: TripleSet -> TripleSet -> String -> Statement
-- construct a statement where the Int indicates the specific graph.
-- command 'st' takes as default graph 0
stf ts1 ts2 s = (ts1,ts2,s)

protost :: String -> Statement
-- transforms a predicate in RDFProlog format to a statement
--             example: "test(a,b,c)."
protost s = st
        where [(cl,_)] = clause s
              (st, _, _) = transClause (cl, 1, 1, 1)

sttopro :: Statement -> String
-- transforms a statement to a predicate in RDFProlog format
sttopro st = toRDFProlog st

sttppro :: Statement -> String
-- transforms a statement to a predicate in RDFProlog format with
-- provenance indication (after the slash)
sttppro st = sttopro st ++ "/" ++ prov st

ants :: Statement -> TripleSet
-- get the antecedents of a statement
ants (ts,_,_) = ts

cons :: Statement -> TripleSet
-- get the consequents of a statement
cons (_,ts,_) = ts

prov :: Statement -> String
-- get the provenance of a statement
prov (_,_,s) = s

fact :: Statement -> TripleSet
-- get the fact = consequents of a statement
fact (_,ts,_) = ts

tvar :: Resource -> Bool
-- test whether this resource is a variable
tvar (Var v) = True
tvar r = False

tlit :: Resource -> Bool
-- test whether this resource is a literal
tlit (Literal l) = True
tlit r = False

turi :: Resource -> Bool
-- test whether this resource is a uri
turi (URI r) = True
turi r = False

grs :: Resource -> String
-- get the string value of this resource
grs r = getRes r

gvar :: Resource -> Vare
```

```
-- get the variable from a resource
gvar (Var v) = v

graph :: TS -> Int -> String -> RDFGraph
-- make a numbered graph from a triple store
-- the string indicates the graph type
-- the predicates 'gtype' and 'gnumber' can be queried
graph ts n s = g4
        where changenr nr (ts1,ts2,_) = (ts1,ts2,intToString nr)
              g1 = getRDFGraph (map (changenr n) ts) s
              g2 = apred g1 ("gtype(" ++ s ++ ").")
              g3@(RDFGraph (array,d,p)) = apred g2 ("gnumber(" ++
                                                    intToString n ++ ").")
              ([],tr,_) = array!1
              array1 = array//[(1,([],tr,intToString n))]
              g4 = RDFGraph (array1,d,p)

agraph :: RDFGraph -> Int -> Array Int RDFGraph -> Array Int RDFGraph
-- add a RDF graph to an array of graphs at position n.
-- If the place is occupied by a graph, it will be overwritten.
-- Limited to 5 graphs at the moment.
agraph g n graphs = graphs//[(n,g)]

-- maxg defines the maximum number of graphs
maxg :: Int
maxg = 3


statg :: RDFGraph -> TS
-- get all statements from a graph
statg g = fromRDFGraph g

pgraph :: RDFGraph -> String
-- print a RDF graph in RDFProlog format
pgraph g = tsToRDFProlog (fromRDFGraph g)

pgraphn3 :: RDFGraph -> String
-- print a RDF graph in Notation 3 format
pgraphn3 g = tsToN3 (fromRDFGraph g)

cgraph :: RDFGraph -> String
-- check a RDF graph. The string contains first the listing of the original
triple
-- store in RDF format and then all statements grouped by predicate. The
two listings
-- must be identical
cgraph g = checkGraph g

apred :: RDFGraph -> String -> RDFGraph
-- add a predicate of arity (length t). g is the rdfgraph, p is the
-- predicate (with ending dot).
apred g p = astg g st
        where st = protost p

astg :: RDFGraph -> Statement -> RDFGraph
-- add statement st to graph g
astg g st = putStatementInGraph g st

dstg :: RDFGraph -> Statement -> RDFGraph
-- delete statement st from the graph g
dstg g st = delStFromGraph g st
```

```
gpred :: RDFGraph -> Resource -> [Statement]
-- get the list of all statements from graph g with predicate p
gpred g p = readPropertySet g p

gpvar :: RDFGraph -> [Statement]
-- get the list of all statements from graph g with a variable predicate
gpvar g = getVariablePredicates g

gpredv :: RDFGraph -> Resource ->[Statement]
-- get the list of all statements from graph g with predicate p
--              and with a variable predicate.
gpredv g p = gpred g p ++ gpvar g

checkst :: RDFGraph -> Statement -> Bool
-- check if the statement st is in the graph
checkst g st
        | f == [] = False
        | otherwise= True
        where list = gpred g (p (head (cons st)))
              f = [st1|st1 <- list, st1 == st]

changest :: RDFGraph -> Statement -> Statement -> RDFGraph
-- change the value of a statement to another statement
changest g st1 st2 = g2
        where list = gpred g (p (head (cons st1)))
              (f:fs) = [st|st <- list, st == st1]
              g1 = dstg g f
              g2 = astg g st2

setparam :: RDFGraph -> String -> RDFGraph
-- set a parameter. The parameter is in RDFProlog format (ending dot
-- included.) . The predicate has only one term.
setparam g param
        | fl == [] = astg g st
        | otherwise = changest g f st
        where st = protost param
              fl = checkparam g param
              (f:fs) = fl

getparam :: RDFGraph -> String -> String
-- get a parameter. The parameter is in RDFProlog format.
-- returns the value of the parameter.
getparam g param
   | fl == [] = ""
   | otherwise = grs (o (head (cons f)))
   where st = protost param
         fl = checkparam g param
         (f:fs) = fl

checkparam :: RDFGraph -> String -> [Statement]
-- check if a parameter is already defined in the graph
checkparam g param
        | ll == [] = []
        | otherwise = [l1]
        where st = protost param
              list = gpred g (p (head (cons st)))
              ll = [l|l <-list, length (cons l) == 1]
              (l1:ll1) = ll

assert :: RDFGraph -> Statement -> RDFGraph
```

```
-- assert a statement
assert g st = astg g st

retract :: RDFGraph -> Statement -> RDFGraph
-- retract a statement
retract g st = dstg g st

asspred :: RDFGraph -> String -> RDFGraph
-- assert a predicate
asspred g p = assert g (protost p)

retpred :: RDFGraph -> String -> RDFGraph
-- retract a statement
retpred g p = retract g (protost p)
```

Fig. 5.3. The mini-language for manipulating triples, statements and **RDF** graphs.

5.4. Languages needed for inferencing

Four languages are needed for inferencing [HAWKE] . These languages are needed for handling:
1) facts
2) rules
3) queries
4) results
Each language is characterised by a vocabulary, a syntax and a semantic interpretation.

Using these languages in inference implies using different operational procedures for each of the languages. This is also the case in **RDFE**ngine. These languages can have (but don't have to) the same syntax but they cannot have the same semantics. The language for facts is based on the **RDF** data model. Though rules have the same representation (they can be represented by triples and thus **RDF** syntax, with variables as an extension), their semantic interpretation is completely different from the interpretation of facts.
The query language presented in this thesis is a rather simple query language, for a standard engine on the World Wide Web something more complex will be needed. Inspiration can be sought in a language like **SQL**.
For representing results in this thesis a very simple format is proposed in 5.9.

5.5. The model interpretation of a rule

A valid **RDF** graph is defined in 2.7. All arcs are labeled with valid **URI**s.

A number of results from the **RDF** model theory [RDFM] are useful .

Subgraph Lemma. A graph implies all its subgraphs.
Instance Lemma. A graph is implied by any of its instances.
An instance of a graph is another graph where one or more blank nodes of the previous graph are replaced by a **URI** or a literal.
Merging Lemma. The merge[1] of a set *S* of **RDF** graphs is implied by *S* and implies every member of *S*.
Monotonicity Lemma. Suppose *S* is a subgraph of *S'* and *S* implies *E*. Then *S'* implies *E*.

These lemmas establish the basic ways of reasoning on a graph. One basic way of reasoning will be added: the implication. The implication, represented in **RDFE**ngine by a rule, is defined as follows:

[1] If graph G1 is represented by triplelist L1 and graph G2 is represented by triplelist L2, then the merge of G1 and G2 is the concatenation of L1 and L2 with elimination of duplicates.

Given the presence of subgraph *SG1* in graph *G* and the implication:
SG1 → *SG2*
the subgraph *SG2* is merged with *G* giving *G'* and if *G* is a valid graph then *G'* is a valid graph too.
Informally, this is the merge of a graph *SG2* with the graph *G* giving *G'* if *SG1* is a subgraph of *G*.

The closure procedure used for axiomatizing **RDFS** in the **RDF** model theory [RDFM] inspired me to give an interpetation of rules based on a closure process.
Take the following rule R describing the transitivity of the 'subClassOf' relation:
{(?c1, subClassOf, ?c2),(?c2, subClassOf, ?c3)} implies {(?c1, subClassOf , ?c3)}.
The rule is applied to all sets of triples in the graph G of the following form:
{(c1, subClassOf, c2), (c2, subClassOf, c3)}
yielding a triple *(c1, subClassOf, c3)*.
This last triple is then added to the graph. This process continues till no more triples can be added. A graph *G'* is obtained called the closure of *G* with respect to the rule set *R*[1]. In the 'subClassOf' example this leads to the transitive closure of the subclass-relation.
If a query is posed: *(cx, subClassOf, cy)* then the answer is positive if this triple is part of the closure *G'*; the answer is negative if it is not part of the closure.
When variables are used in the query:
(?c1, subClassOf, ?c2)
then the solution consists of all triples (subgraphs) in the closure *G'* with the predicate 'subClassOf'.
The process of matching the antecedents of the rule *R* with a graph *G* and then adding the consequent to the graph is called *applying* the rule *R* to the graph *G*.

The process is illustrated in fig. 5.4 and 5.5 with a similar example.

When more than one rule is used the closure *G'* will be obtained by the repeated application of the set of rules till no more triples are produced.
Any solution obtained by a resolution process is then a valid solution if it is a subgraph of the closure obtained.
I will not enter into a detailed comparison with first order resolution theory but I want to mention following point:
the closure graph is the equivalent of a minimal Herbrand model in first order resolution theory or a least fixpoint in fixpoint semantics.[VAN BENTHEM]

[1] In the example a singleton set

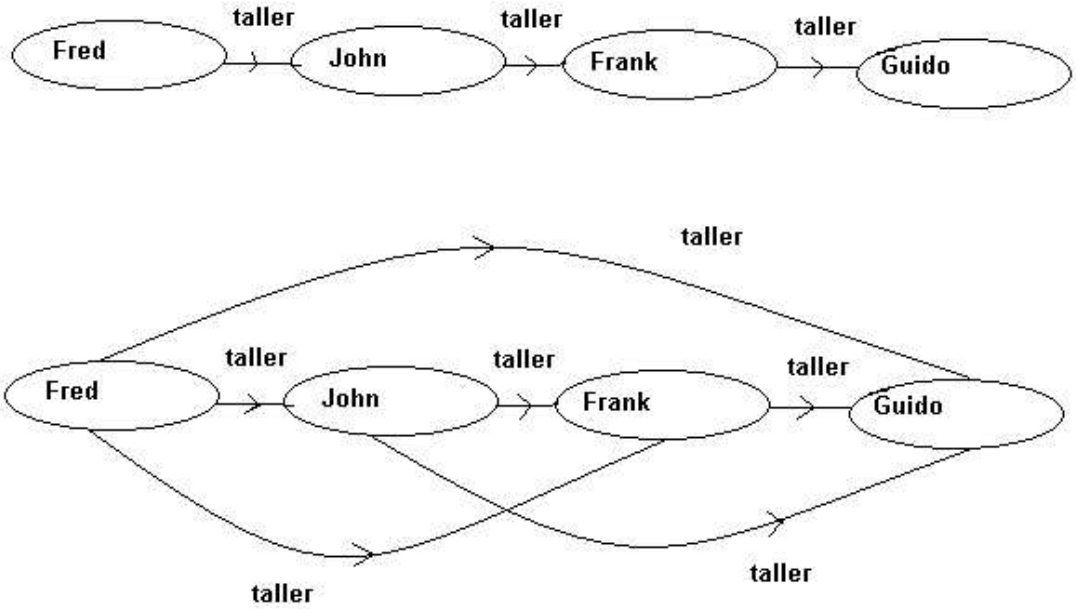

Fig.5.4. A graph (above) with three triples with the *taller relation* and a graph (below) representing the *transitive closure* of the first graph with respect to the relation taller.

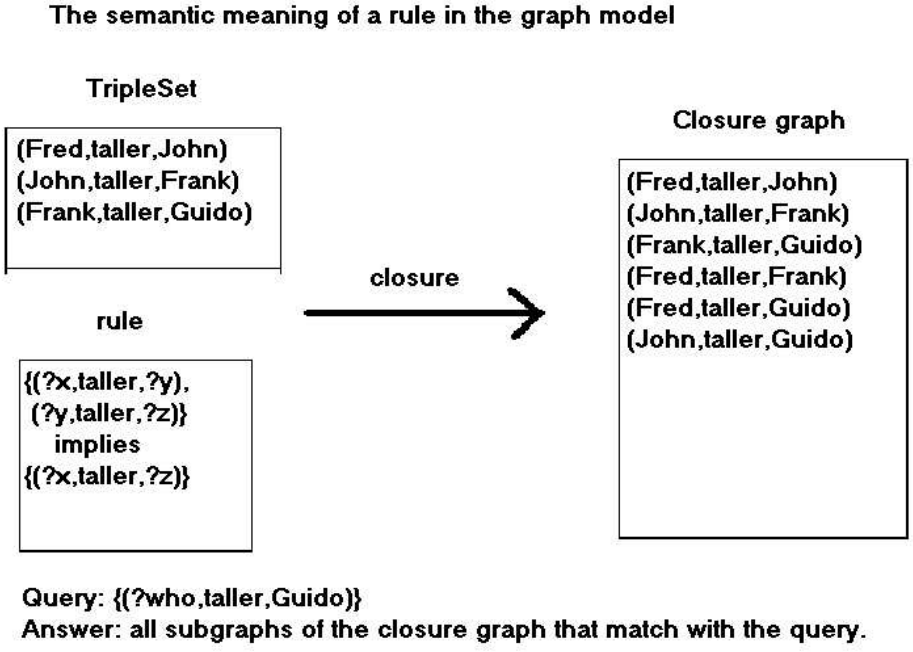

Fig. 5.5. The closure *G'* of a graph *G* with respect to the given rule set. The possible answers to the query are: *(Frank,taller,Guido), (Fred,taller,Guido), (John,taller,Guido).*

5.6. Unification

5.6.1. Example

Fig.5.6 gives a schematic view of multiple matches of a triple with a triplelist. The triple *(Bart, son, ?w)* unifies with two triples *(Bart, son, Jan)* and *(Bart, son, Michèle)*. The triple *(?w, son, Paul)* matches with *(Guido, son, Paul)* and *(Jan, son, Paul)*. This gives 4 possible combinations of matching triples. Following the theory of resolutions all those combinations must be contradicted. In graph terms: they have to match with the closure graph. However, the common variable *?w* between the first and the second triple of triple set 1 is a restriction. The object of the first triple must be the same as the subject of the second triple with as a consequence that there is one solution: *?w = Jan*. Suppose the second triple was: *(?w1, son, Paul)*. Then this restriction does not any more apply and there are 4 solutions:
{(Bart, son, Jan),(Guido, son, Paul)}
{(Bart, son, Jan), (Jan, son, Paul)}
{(Bart, son, Michèle), (Guido, son, Paul)}
{(Bart, son, Michèle), (Jan, son, Paul)}

This example shows that when unifying two triplelists the product of all the alternatives has to be taken while the substitution has to be propagated from one triple to the next for the variables that are equal and also within a triple if the same variable or node occurs twice in the same triple.

5.6.2. Description

In the classic theory of resolution, if θ is a substitution and A a term, then θA represents the term that exists after applying the substitution θ to A.
Two terms are then *unifiable* when there is a substitution θ such that: $\theta A = \theta B$.
Example: *P(x, f(y))* and *P(a, f(g(z)))* are unifiable by $\theta = [(a, x), (g(z),y)]$.
Whether two terms are unifiable can always be decided. In that case it is possible to find a *most general unifying* substitution [VANBENTHEM].
In **RDF** the situation is more simple.
Let *(s1, p1, o1)* and *(s2, p2, o2)* be two triples.
The two triples unify when *s1* unifies with *s2*, *p1* unifies with *p2* and *o1* unifies with *o2*. Because the terms are simple there is only one possible substitution.

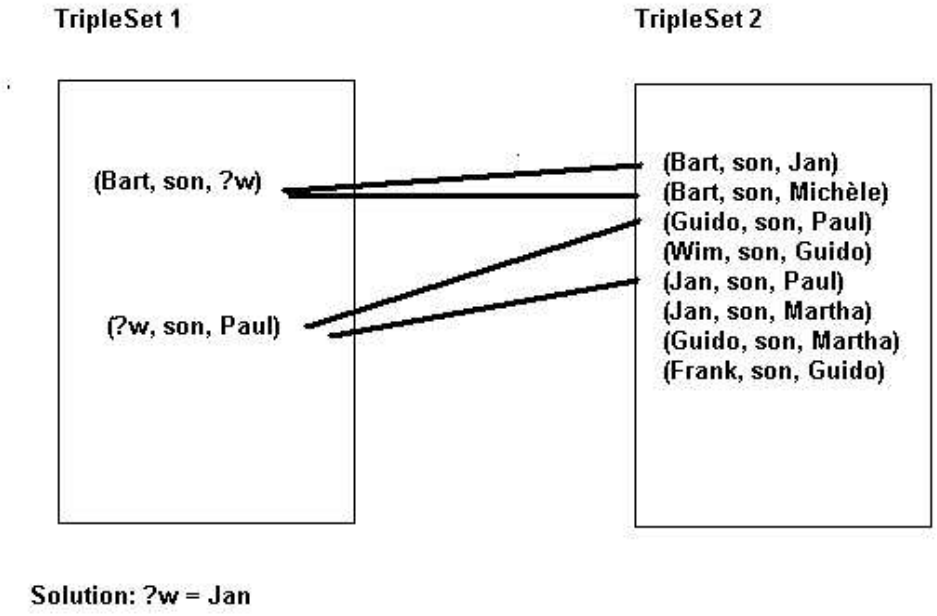

Fig.5.6. Multiple matches within one triplelist.

If an element s_x is a variable, then it unifies with an element s_y that is a **URI** or a literal[1]. Two **URI**'s unify if they are the same. The result of a unification is a substitution. A substitution is a list of tuples (variable, **URI** or Literal). The list can be empty. The result of the unification of two triples is a substitution (containing at most 3 tuples). Applying a substitution to a variable in a triple means replacing the variable with the **URI** or literal defined in the corresponding tuple.

In Haskell:

type ElemSubst = (Vare, Resource)
type Subst = [ElemSubst]
A nil substitution is an empty list:
nilsub :: Subst
nilsub = []

The unification of two statements is given in fig. 5.7.
The application of a substitution is given in fig. 5.8.

```
-- appESubstR applies an elementary substitution to a resource
appESubstR :: ElemSubst -> Resource -> Resource
appESubstR (v1, r) v2
```

[1] or a triplelist, but this is not used in RDFEngine.

```
   | (tvar v2) && (v1 == gvar v2) = r
   | otherwise = v2

-- applyESubstT applies an elementary substitution to a triple
appESubstT :: ElemSubst -> Triple -> Triple
appESubstT es TripleNil = TripleNil
appESubstT es t =
   Triple (appESubstR es (s t), appESubstR es (p t), appESubstR es (o t))

-- appESubstSt applies an elementary substitution to a statement
appESubstSt :: ElemSubst -> Statement -> Statement
appESubstSt es st = (map (appESubstT es) (ants st),
                                map (appESubstT es) (cons st), prov st)

-- appSubst applies a substitution to a statement
appSubst :: Subst -> Statement -> Statement
appSubst [] st = st
appSubst (x:xs) st = appSubst xs (appESubstSt x st)

 -- appSubstTS applies a substitution to a triple store
appSubstTS :: Subst -> TS -> TS
appSubstTS subst ts = map (appSubst subst) ts
```

Fig.5.7. The application of a substitution. The Haskell source code uses the mini-language defined in **RDFML**.hs

In the application of a substitution the variables in the data structures are replaced by resources whenever appropriate.

5.7. Matching two statements

The Haskell source code in fig. 5.8. gives the unification functions.

```
--- Unification:

-- unifyResource r1 r2 returns a list containing a single substitution s
which is
--             the most general unifier of terms r1 r2.  If no unifier
--             exists, the list returned is empty.


unifyResource :: Resource -> Resource -> Maybe Subst
unifyResource r1 r2
      |(tvar r1) && (tvar r2) && r1 == r2 =
                                      Just [(gvar r1, r2)]
        | tvar r1 = Just [(gvar r1, r2)]
        | tvar r2 = Just [(gvar r2, r1)]
        | r1 == r2 = Just []
      | otherwise = Nothing

unifyTsTs :: TripleSet -> TripleSet -> Maybe Subst
unifyTsTs ts1 ts2
        | res == [] = Nothing
```

```
        | otherwise = Just (concat res1)
        where res = elimNothing
                      ([unifyTriples t1 t2|t1 <- ts1, t2 <- ts2])
              res1 = [sub |Just sub <- res]


elimNothing [] = []
elimNothing (x:xs)
      | x== Nothing = elimNothing xs
      | otherwise = x:elimNothing xs

unifyTriples :: Triple -> Triple -> Maybe Subst
unifyTriples TripleNil TripleNil = Nothing
unifyTriples t1 t2 = -- error (show sub1 ++ show sub2 ++ show sub3)
      if (subst1 == Nothing) || (subst2 == Nothing)
                             || (subst3 == Nothing) then Nothing
           else Just (sub1 ++ sub2 ++ sub3)
      where subst1 = unifyResource (s t1) (s t2)
            subst2 = unifyResource (p t1) (p t2)
            subst3 = unifyResource (o t1) (o t2)
            Just sub1 = subst1
            Just sub2 = subst2
            Just sub3 = subst3

-- unify two statements
unify :: Statement -> Statement -> Maybe Match
unify st1@(_,_,s1) st2@(_,_,s2)
      | subst == Nothing = Nothing
      | (trule st1) && (trule st2) = Nothing
      | trule st1 = Just (Match subst1 (transTsSts (ants st1) s1)
[(st2,st1)])
      | trule st2  = Just (Match subst1 (transTsSts (ants st2) s2)
[(st2,st1)])
      | otherwise = Just (Match subst1 [stnil] [(st2,st1)])
      where subst = unifyTsTs (cons st1)(cons st2)
            Just subst1 = subst
```

Fig. 5.8. The unification of two statements

When matching a statement with a rule, the consequents of the rule are matched with the statement. This gives a substitution. This substitution is returned together with the antecedents of the rule and a tuple formed by the statement and the rule. The tuple is used for a history of the unifications and will serve to produce a proof when a solution has been found.
The antecedents of the rule will form new goals to be proven in the inference process.

An example:

Facts:
(Wim, brother, Frank)
(Christine, mother, Wim)
Rule:
[(?x, brother, ?y), (?z, mother, ?x)] implies [(?z, mother, ?y)]
Query:
[(?who, mother, Frank)]
The unification of the query with the rule gives the substitution:
[(?who, ?z), (?y, Frank)]
Following new facts have then to be proven:
(?x, brother, Frank), (?who, mother, ?x)
(?x, brother, Frank) is unified with (Wim, brother, Frank) giving the substitution:
[(?x, Wim)]
and:
(?who, mother, ?x) is unified with *(Christine, mother, Wim)* giving the substitution:
[(?who, Christine), (?x, Wim)]
The solution then is:
(Christine, mother, Frank).

5.8. The resolution process

RDFEngine has been built with a standard **SLD** inference engine as a model: the Prolog engine included in the Hugs distribution [JONES]. The basic backtracking mechanism of **RDFE**ngine is the same. A resolution based inference process has three possible outcomes:
1) it ends giving one or more solutions
2) it ends without a solution
3) it does not end (looping state)
An anti-looping mechanism has been provided in **RDFE**ngine. This is the mechanism of De Roo [DEROO].
The resolution process starts with a query. A query is a triplelist. The triples of the query are entered into the goallist. The goallist is a list of triples. A goal is selected and unified with a **RDF** graph. A goal unifies with a rule when it unifies with the consequents of the rule. The result of the unification with a rule is a substitution and a list of goals.
A goal can unify with many other facts or rules. Each unification will generate an *alternative*. An alternative consists of a substitution and a list of goals. Each alternative will generate new alternatives in the inference process. The graph representing all alternatives in sequence from the start till the end of the

inference process has a tree structure[1]. A path from a node to a leaf is called a *search path*.
When a goal does not unify with a fact or a rule, the search path that was being followed ends with a *failure.* A search path ends with a leaf when the goallist is empty. At that point all accumulated substitutions form a solution i.e. when these substitutions are applied to the query, the query will become grounded.
A *looping search path* is an infinite search path. An anti-looping mechanism is necessary because, if there is not one, many queries will produce infinite search paths.
When several alternatives are produced after a unification, one of the alternatives will be investigated further and the others will be pushed on a *stack*, together with the current substitution and the current goallist. An entry in the stack constitutes a *choicepoint.* When a search path ends in a failure, a *backtrack* is done to the latest choicepoint. The goallist and the current substitution are retrieved from the stack together with the set of alternatives. One of the alternatives is chosen as the start of a new search path.
RDFEngine uses a depth first search. This means that a specific search path is followed till the end and then a backtrack is done.
In a breadth first search for all paths one unification is done, one path after another. All paths are thus followed at the same time till, one by one, they reach an endpoint. In a breadth first search also an anti-looping mechanism is necessary.

5.9. Structure of the engine

5.9.1. Introduction

The *inference engine* : this is where the resolution is executed. In **RDFE**ngine the basic **RDF** data is contained in *an array of RDF graphs*. The inferencing is done using all graphs in the array.
It is possible to define the resolution process as a finite state machine. Fig. 5.9. gives an overview.

[1] Each node generates zero or more new nodes.

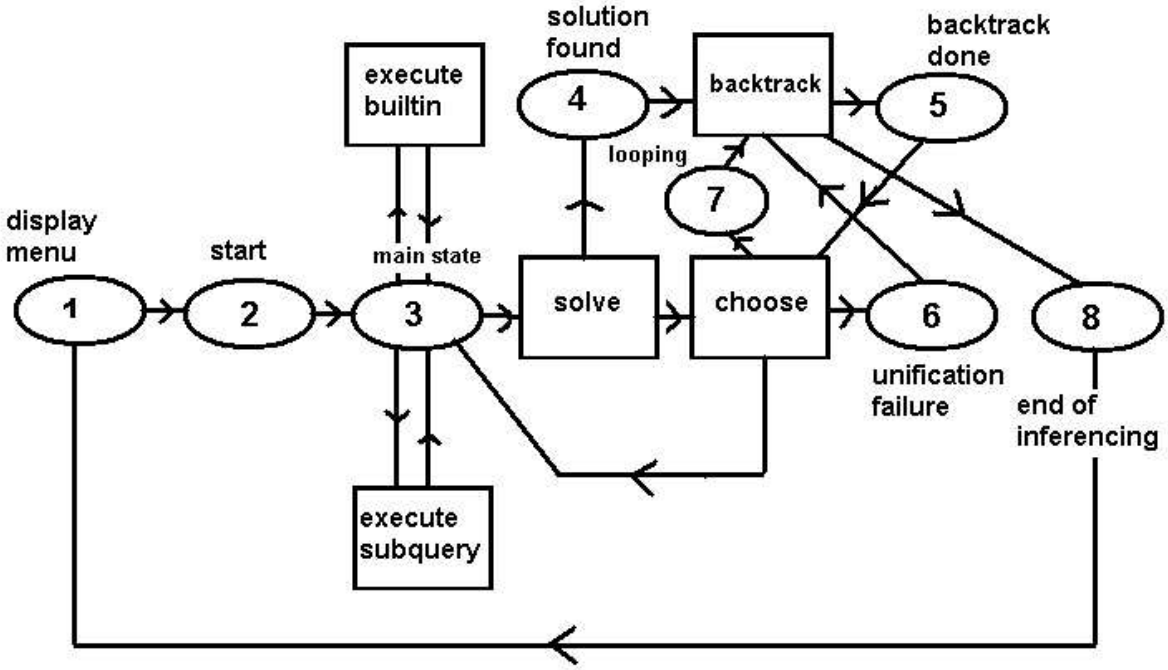

Fig. 5.9. The inference process as a Finite State Machine (FSM).

5.9.2. Detail description

The ovals in fig.5.9 indicate the states; the rectangles are functions, that are executed when passing from state to another. **FSM** is the abbreviation of Finite State Machine. Here follows an overview of all state changes.

1 → 2
The proces starts with state 1 where the menu is displayed. Besides functions that are not shown in the figure, from the menu an inferencing process can be started by selecting a database and a query.

2 → 3
In state 2 the data structures for the inferencing process are prepared and the query is entered in the goallist. Then the main state, 3, of the process is entered.

3 → 3
Depending on the next triple in the goallist there are three possibilities:

a) the triple contains a builtin e.g. (Txxx, io:print, ”Message to the user”). Some following triples might also be used. After executing the builtin, the **FSM** will return to state 3.
b) The triple and a certain number of following triples define a subquery. The subquery is directed to another server. The returned results are integrated into the database and the **FSM** returns to state 3.

c) In the function *solve* the first goal in the goallist is selected and unified with the database, giving a list of alternatives. In the function *choose* one of the alternatives is selected and added to the goallist. The other alternatives are pushed on the stack. The **FSM** returns to state 3.

3 → 4
The solve function detects an empty goallist. This means all goals have been 'proved' and the resolution refutation process is complete. A solution has been found and the **FSM** passes to state 4.

4 → 5
A solution has been found. The **FSM** passes control to the *backtrack* function that looks if alternatives are present on the stack for further investigation. If this is the case, then control is passed to state 5.

3 → 6
The solve function tries to unify a goal with the database; however the unification process fails. The choose function detects the lack of alternatives and the **FSM** passes to state 6.

3 → 7
A looping has been detected in function choose. The **FSM** passes control to state 6. Looping and the anti-looping mechanism will be discussed in 5.14.

6 → 5
7 → 5
The current search path is aborted. The backtrack function looks for further alternatives; if this is the case, the **FSM** passes control to state 5.

5 → 3
A backtrack has been done. An alternative will be chosen by the function choose.

4 → 8
6 → 8
No more alternatives are present on the stack. The inferencing process is finished. The **FSM** passes to state 8.

8 → 1
The inferencing process is finished. Results are displayed and/or used and control is passed to state 1.

Definition: an *inference step* is the act of passing from one state to another (with the exception of states 1 and 2).
An inference step uses a data structure *InfData*:

```
data InfData = Inf {graphs::Array Int RDFGraph, goals::Goals, stack::Stack,
                    lev::Level, pdata::PathData, subst::Subst,
                    sols::[Solution], ml::MatchList, mes::String }
```

This data structure is specified in an Abstract Data Structure Inf**ADT**.hs (fig.5.10). It illustrates how complex data structures can easily be handled in Haskell using *field labels.* The use of field labels provides for *evolvability* of the datastructure because single fields can be easily removed or added without provoking major changes to the program. For instance, to get access to the stack, the instruction:
stackI = stack inf where *inf* is an instance of *InfData,* can be used.
This is a complex data structure. Its acess is further simplified by access routines in the **ADT** like e.g. *sstack*: show the contents of the stack in string format.
The elements of the data structure are :
Array Int RDFGraph: The array containing the **RDF** graphs.
Goals: The list of goals that have to be proved.
Stack: The stack needed for backtracking.
Level: The inferencing level. This is needed for renumbering the variables and backtracking.
PathData: A list of the statements that have been unified. This is used to establish the proof of the solutions.
Subst: The current substitution. This substitution must be applied to the current goal.
[Solution]: The list of the solutions.
MatchList: This is a list of *Match* constructors:

```
data Match = Match {msub::Subst, mgls::Goals, bs::BackStep}
```

Matching two statements gives a substitution, a list of goals (that may be empty) and a *backstep*. A backstep is nothing else than a tuple formed by the two matched statements. This is needed for producing the proof of a solution.
A substitution with a list of goals forms an alternative.
String: This string is used for maintaining the state of the process.

An inference step has as only input an instantiation of InfData and as only output an instantiation of InfData. The data structure contains all the necessary information for the next inference step.
Another mini-language for inferencing is defined in module Inf**ML**.hs.
It is shown in fig. 5.11.
Fig. 5.12 gives the source code of the module that executes the inferencing process **RDFE**ngine.hs.

```
data InfData = Inf {graphs::Array Int RDFGraph, goals::Goals, stack::Stack,
                    lev::Level, pdata::PathData, subst::Subst,
                    sols::[Solution], ml::MatchList, mes::String}

* Array Int RDFGraph: this is the array of RDF graphs
* Goals: this is the list of goals. The format is:
              type Goals = [Statement]
* Stack: the stack contains all data necessary for backtracking. The format
is:
             data StackEntry = StE {sub::Subst, fs::FactSet,
sml::MatchList}
             type Stack = [StackEntry]
The stack is a list of triples. The first element in the triple is a
substitution. The second element is a list of facts:
             type FactSet = [Statement]
The third element is a list of matches. A match is a triple and is the
result of matching two statements:
              type MatchList = [Match]
             data Match = Match {msub::Subst, mgls::Goals, bs::BackStep}
Matching two statements gives a substitution, a list of goals (that may be
empty)  and a  backstep. A backstep is nothing else than a tuple formed by
the two matched statements:
            type BackStep = (Statement, Statement)
The purpose of the backstep is to keep a history of the unifications. This
will serve for constructing the proof of a solution when it has been found.
* Level: an integer indicating the backtacking level.
* PathData: this is a list of tuples.
           data PDE = PDE {pbs::BackStep, plev::Level}
           type PathData = [PDE]
 The pathdata forms the history of the occurred unifications. The level is
kept also for backtracking.
* Subst: the current substitution
* [Solution]: the list of the solutions. A solution is a tuple:
           data Solution = Sol {ssub::Subst, cl::Closure}
A solution is formed by a substitution and a closure. A closure is a list
of forward steps. A forward step is a tuple of statements. The closure is
the proof of the solution. It indicates how the result can be obtained with
forward reasoning. It contains the statements and the rules that have to be
applied to them.
                    type ForStep = (Statement,Statement)
                    type Closure = [ForStep]
* MatchList: the type is explained above. The function choose needs this
parameter for selecting new goals.
* String: this string indicates the state of the ifnerence process.
```

5.10. The data structure needed for performing an inference step.

```
Description of the Mini Language for inferencing
For the Haskell data types: see RDFData.hs

esub :: Vare -> Resource -> ElemSubst (esub = elementary substitution)
esub v t : define an elementary substitution

apps :: Subst -> Statement -> Statement (apps = apply susbtitution)
apps s st : apply a susbtitution to a statement

appst :: Subst -> Triple -> Triple (appst = apply substitution (to) triple)
appst s t : apply a substitution to a triple

appsts :: Substitution -> TripleSet (appsts = apply substitution (to)
tripleset)
appsts s ts : apply a substitution to a tripleset

unifsts :: Statement -> Statement -> Maybe Match (unifsts = unify
statements)
unifsts st1 st2 : unify two statements giving a match
                        a match is composed of: a substitution,
                        the returned goals or [], and a backstep.

gbstep :: Match -> BackStep (gbstep = get backwards (reasoning) step
gbstep m : get the backwards reasoning step (backstep) associated with a
match.
                 a backstep is composed of statement1 and statement 2 of
the match
               (see unifsts)

convbl :: [BackStep] -> Substitution -> Closure (convbl = convert a
backstep list)
convbl bl sub : convert a list of backsteps to a closure using the
substitution sub

gls :: Match -> [Statement] (gls = goals)
gls : get the goals associated with a match

gsub :: Match -> Substitution (gsub = get substitution)
gsub m : get the substitution from a match
```

Fig. 5.11. The mini-language for inferencing

A process of *forward reasoning* applies rules to **RDF** facts deducing new facts. These facts are added to the **RDF** graph. A solution is found when a graph matching of the query with the **RDF** graph is possible. Either the process continues till one solution has been found, or it continues till no more facts can be deduced.
When **RDFE**ngine finds a solution, the function *convbl* converts a list of backsteps to a *closure*. It thereby uses a substitution that is the concatenated, accumulated substitution associated with the search path that leads to the solution. The closure is the proof of the solution. It gives all the steps that are necessary to deduce the solution using a forward reasoning process. The correctness of the function convbl will be proved later.

```
type BackStep = (Statement, Statement)
type Closure = [ForStep]
type ForStep = (Statement, Statement)
convbl :: [BackStep] -> Subst -> Closure
-- convert a list of backsteps to a closure using the substitution sub
convbl bsl sub = asubst fcl
      where fcl = reverse bsl
            asubst [] = []
            asubst ((st1,st2):cs) = (appSubst sub st1, appSubst sub st2):
                                    asubst cs
```

```
-- RDFEngine version of the engine that delivers a proof
-- together with a solution

solve inf
        | goals1 == [] = inf{mes="solution"}
        | bool = solve inf2
        | otherwise =  choose inf1
        where goals1 = goals inf
              (g:gs) = goals1
              bool = g == ([],[],"0")
              level = lev inf
              graphs1 = graphs inf
              s = subst inf
              matchlist = alts graphs1 level (apps s g)
              inf1 = inf{goals=gs, ml=matchlist}
              inf2 = inf{goals=gs}

backtrack inf
      | stack1 == [] = inf{mes="endinf"}
      | otherwise = inf1{mes="backtracked"}
      where stack1 = stack inf
            ((StE subst1 gs ms):sts) = stack1
            pathdata = pdata inf
            level = lev inf
            newpdata = reduce pathdata (level-1)

            inf1 = inf{stack=sts,
```

```
                        pdata=newpdata,
                        lev=(level-1),
                        goals=gs,
                        subst=subst1,
                        ml=ms}

choose inf
      | matchlist == [] = inf{mes="failure"}
      -- anti-looping controle
      | ba ‘stepIn‘ pathdata = inf{mes="looping"}
      | otherwise = inf1{mes="main"}
      where matchlist = ml inf
            (Match subst1 fs [ba]):ms = matchlist
            subst2 = subst inf
            gs = goals inf
            stack1 = (StE subst2 gs ms):stack inf
            pathdata = pdata inf
            level = lev inf
            newpdata = (PDE [ba] level):pathdata
            mes1 = mes inf

            inf1 = inf{stack=stack1,
                       lev=(level+1),
                       pdata=newpdata,
                       subst=subst2 ++ subst1,
                       goals = fs ++ gs}

-- get the alternatives, the substitutions and the backwards rule
applications.
-- getMatching gets all the statements with the same property or
-- a variable property.
alts :: Array Int RDFGraph -> Level -> Fact -> MatchList
alts ts n g = matching (renameSet (getMatching ts g) n) g
```

Fig. 5.12. The inference engine. The source code uses the mini-languages defined in **RDFML**.hs, Inf**ML**.hs and **RDFD**ata.hs. The constants are mainly defined in **RDFD**ata.hs.

5.10. The closure path

5.10.1. Problem

In the following paragraphs I develop a notation that enables to make a comparison between forward reasoning (the making of a closure) and backwards reasoning (the resolution process). The meaning of a rule and what represents a solution to a query are defined using forward reasoning. Therefore the properties of the backward reasoning process are proved by comparing with the forward reasoning process. This will serve to proof the soundness, the completeness and the monotonicity of **RDFE**ngine[1] and the correctness of the anti-looping mechanism. It will also indicate how a proof of solutions can be obtained and how procedures can be generated from a proof (chapter 6).
Note: closure paths and resolution paths are not paths in an **RDF** graph but in a search tree.

5.10.2. The closure process

The original graph that is given will be called *G*. The closure graph resulting from the application of the set of rules will be called *G'* (see further). A rule will be indicated by a miniscule with an index. A subgraph of *G* or another graph (query or other) will be indicated by a miniscule with an index.
A rule is applied to a graph *G*; the antecedents of the rule are matched with one or more subgraphs of *G* to yield a consequent that is added to the graph *G*. The consequent must be grounded for the graph to stay a valid **RDF** graph. This means that the consequent can not contain variables that are not in the antecedents.

5.10.3. Notation

A *closure path* is a finite sequence of triples (not to be confused with **RDF** triples):
$(g_1, r_1, c_1), (g_2, r_2, c_2), …., (g_n, r_n, c_n)$
where (g_1, r_1, c_1) means: the rule r_1 is applied to the subgraph (tripleset) g_1 to produce the subgraph (tripleset) c_1 that is added to the graph. A closure path represents a part of the closure process. This is not necessarily the full closure process though the full process is represented also by a closure path.
The *indices* in g_i, r_i, c_i represent sequence numbers but not the identification of the rules and subgraphs. So e.g. r_7 could be the same rule as r_{11}.I introduce this notation to be able to compare closure paths and resolution paths (defined

[1] Builtins can, when added to RDFEngine, break the monotonicity.

further) for which purpose I do not need to know the identification of the rules and subgraphs.
It is important to remark that all triples in the triplesets (subgraphs) are grounded i.e. do not contain variables. This imposes the restriction on the rules that the consequents may not contain variables that are absent in the antecedents.

5.10.4. Examples

In fig.5.14 rule 1 *{(?x, taller, ?y),(?y, taller, ?z)} implies {(?x, taller, ?z)}*
matches with the subgraph *{(John, taller, Frank),(Frank, taller, Guido)}*
to produce the subgraph *{(John, taller, Guido)}* and the substitution *((?x, John), (?y, Frank),(?z, Guido))*
In the closure process the subgraph *{(John, taller, Guido)}* is added to to the original graph.

Fig. 5.13 , 5.14 and 5.15 illustrate the closure paths.

In fig. 5.14 the application of rule 1 and then rule 2 gives the closure path (I do not write completely the rules for brevity):
({(John, taller, Frank),(Frank, taller, Guido)}, rule 1, {(John, taller, Guido)}), ({(John, taller, Guido)}, rule 2, {(Guido, shorter, John)})
Two triples are added to produce the closure graph:
(John, taller, Guido) and *(Guido, shorter, John)*.

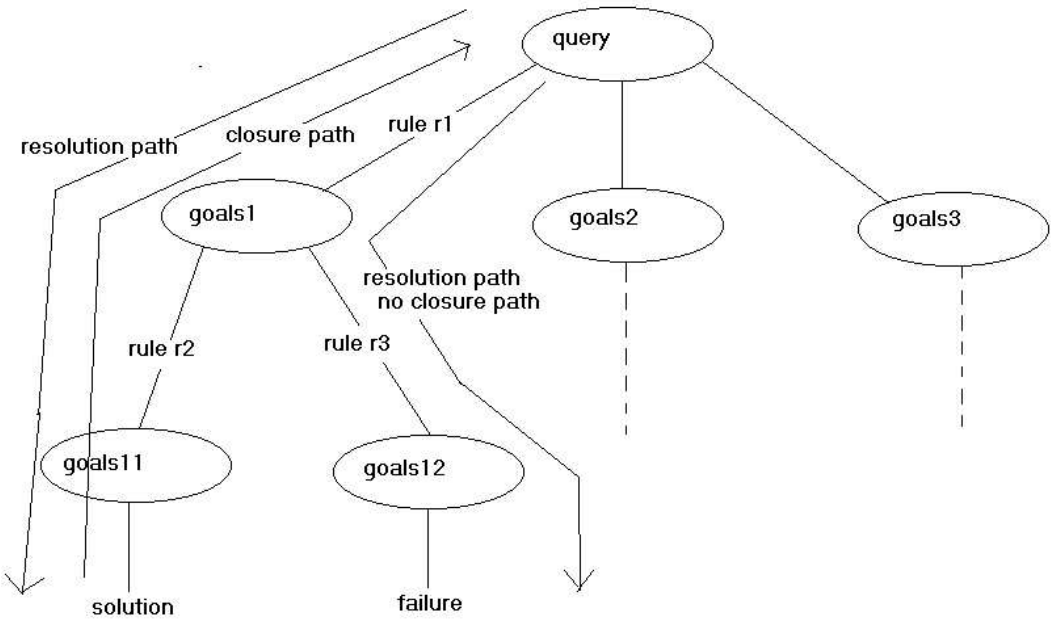

Fig. 5.13.The resolution process with indication of resolution and closure paths.

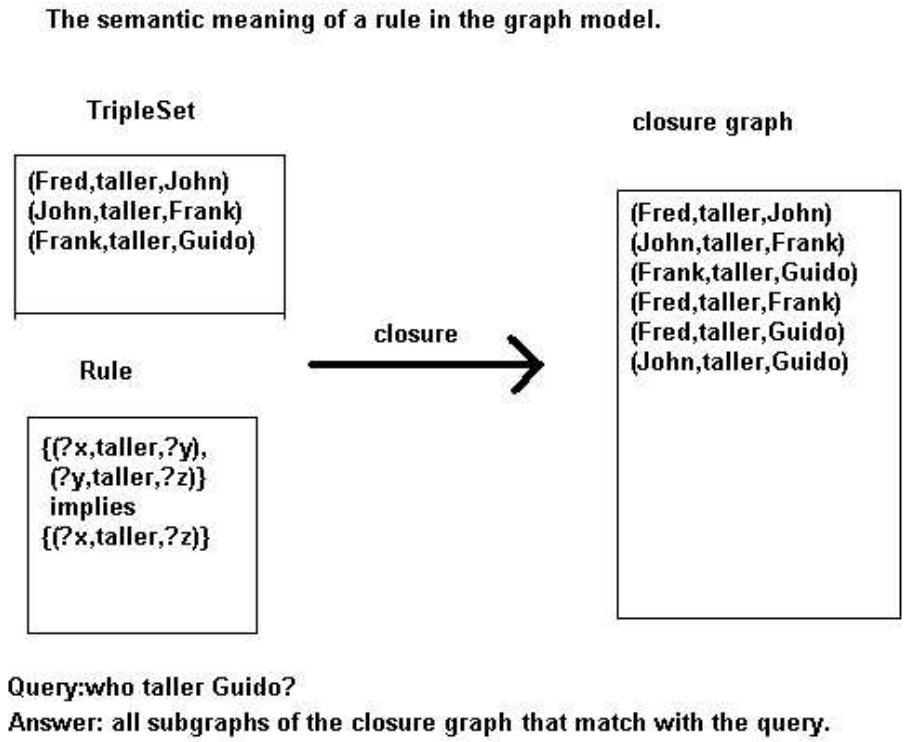

Fig.5.14. The closure of a graph G

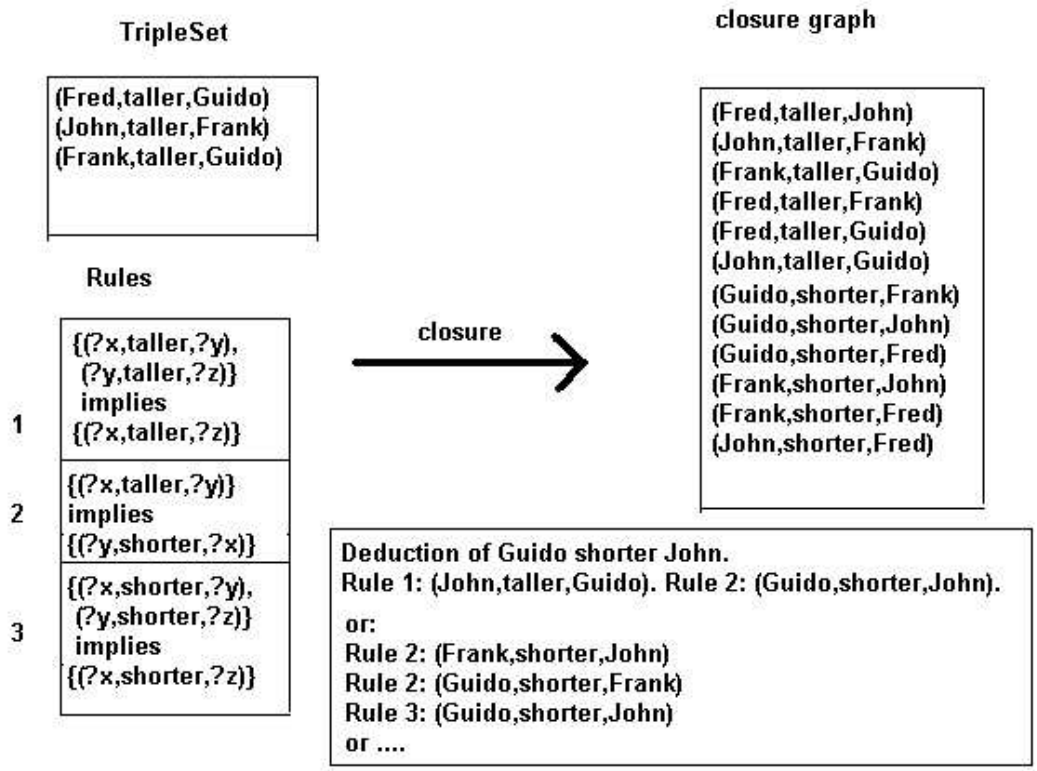

Fig.5.15. Two ways to deduce the triple (Guido,shorter,John).

5.11. The resolution path

5.11.1. Problem

The reason for describing a resolution path was already explained in 5.10.1.

5.11.2. The process

In the resolution process the query is unified with the database giving alternatives. An alternative is a tripleset and a substitutionlist. One tripleset becomes a goal. This goal is then unified etc… When a unification fails a backtrack is done and the process restarts with a previous goal. This gives in a resolution path a series of goals: query, goal1, goal2,… each goal being a subgraph. If the last goal is empty then a solution has been found; the solution is a substitutionlist applied to the query. When this substitutionlist is applied to the series of goals: query, goal1, goal2, … of the *resolution path* all subgraphs in the resolution path will be grounded i.e. they will not any more contain any variables (otherwise the last goal cannot be empty). The reverse of such a grounded resolution path is a *closure path*. I will come back on this. This is illustrated in fig. 5.13.

A subgraph *g1* (a goal) unifies with the consequents of a rule with result a substitution *s* and a subgraph *g1'* consisting of the antecedents of the rule

or
g1 is unified with a connected subgraph of *G* with result a substitution *s1* and no subgraph in return. In this case I will assume *an identity rule* has been applied i.e. a rule that transforms a subgraph into the same subgraph with the variables instantiated. An identity rule is indicated with r_{id}.

The application of a rule in the resolution process is the reverse of the application in the closure process. In the closure process the antecedents of the rules are matched with a subgraph and replaced with the consequents; in the resolution process the consequents of the rule are matched with a subgraph and replaced with the antecedents.

5.11.3. Notation

A *resolution path* is a finite sequence of triples (not to be confused with **RDF** triples):
$(c_1, r_1, g_1), (c_2, r_2, g_2), \ldots,(c_n, r_n, g_n)$
(c_i, r_i, g_i) means: a rule r_i is applied to subgraph c_i to produce subgraph g_i.
Contrary to what happens in a closure path g_i can still contain variables.
Associated with each triple (c_i, r_i, g_i) is a substitution s_i.
The accumulated list of substitutions $[s_i]$ must be applied to all triplesets that are part of the path. This is very important.

5.11.4. Example

An example from fig.5.16:
The application of *rule 2* to the query *{(?who, shorter, John)}* gives the substitution *((?x, John),(?y, Frank))* and the antecedents *{(Frank, shorter, John)}*.
An example of closure path and resolution path as they were produced by a trace of the engine can be found in appendix 2.

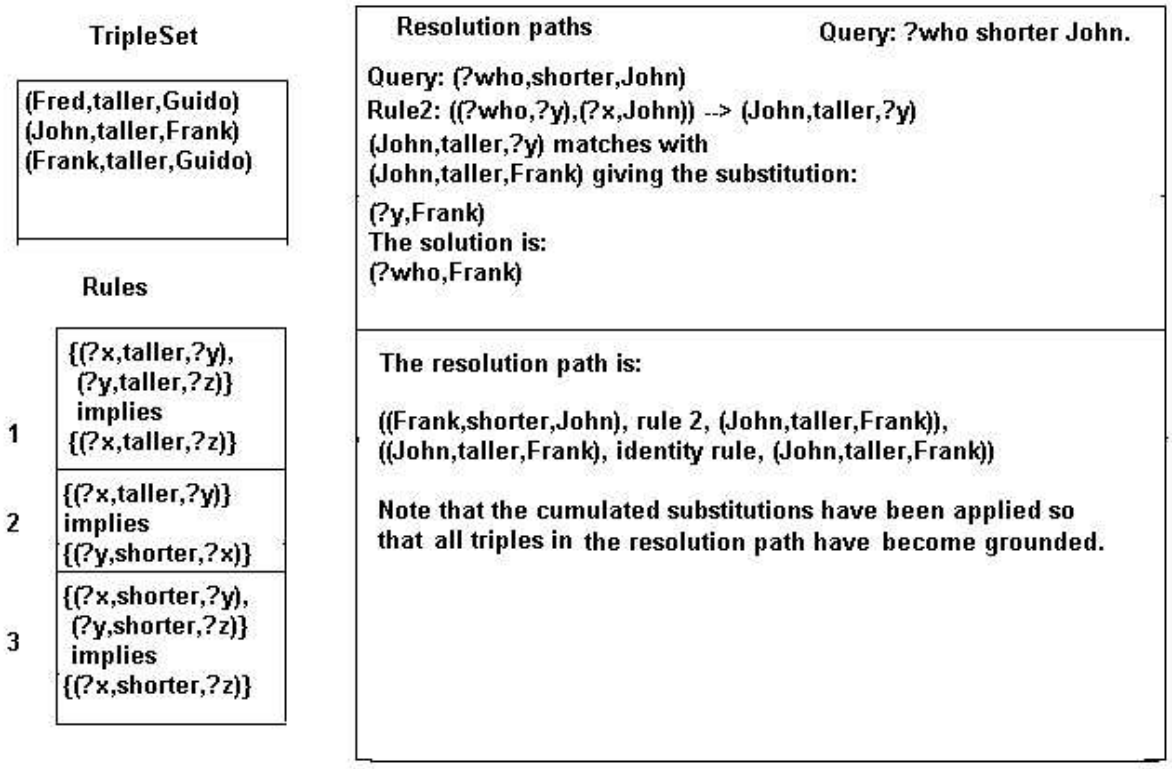

Fig.5.16. Example of a resolution path.

5.12. Variable renaming

There are two reasons why variables have to be renamed when applying the resolution process (fig.5.17) [AIT]:

1) Different names for the variables in different rules. Though the syntaxis permits to define the rules using the same variables the inference process cannot work with this. Rule one might give a substitution *(?x, Frank)* and later on in the resolution process rule two might give a substitution *(?x, John)*. So by what has *?x* to be replaced?
   It is possible to program the resolution process without this renaming but I think it's a lot more complicated than renaming the variables in each rule.
2) Take the rule:
   [(?x1, taller, ?y1),(?y1, taller, ?z1)] implies [(?x1, taller, ?z1)]
   At one moment *(John, taller, Fred)* matches with *(?x1, taller, ?z1)* giving substitutions:
   ((?x1, John),(?z1, Fred)).
   Later in the process *(Fred, taller, Guido)* matches with *(?x1, taller, ?z1)* giving substitutions:
   ((?x1, Fred),(?z1, Guido)).

So now, by what is *?x1* replaced? Or *?z1*?
This has as a consequence that the variables must receive a unique name what is done by adding the *level number* to the name where each step in the resolution process is attributed a level number.
The substitutions above then become e.g.
((1_?x1, John),(1_?x1, Fred))
and
((2_?x1 , Fred),(2_?z1, Guido))
This numbering is mostly done by prefixing with a number as done above. The variables can be renumbered in the database but it is more efficient to do it on a copy of the rule before the unification process.
In some cases this renumbering has to be undone e.g. when comparing goals.
The goals:
(Fred, taller, 1_?x1).
(Fred, taller, 2_?x1).
are really the same goals: they represent the same query: all people shorter than Fred.

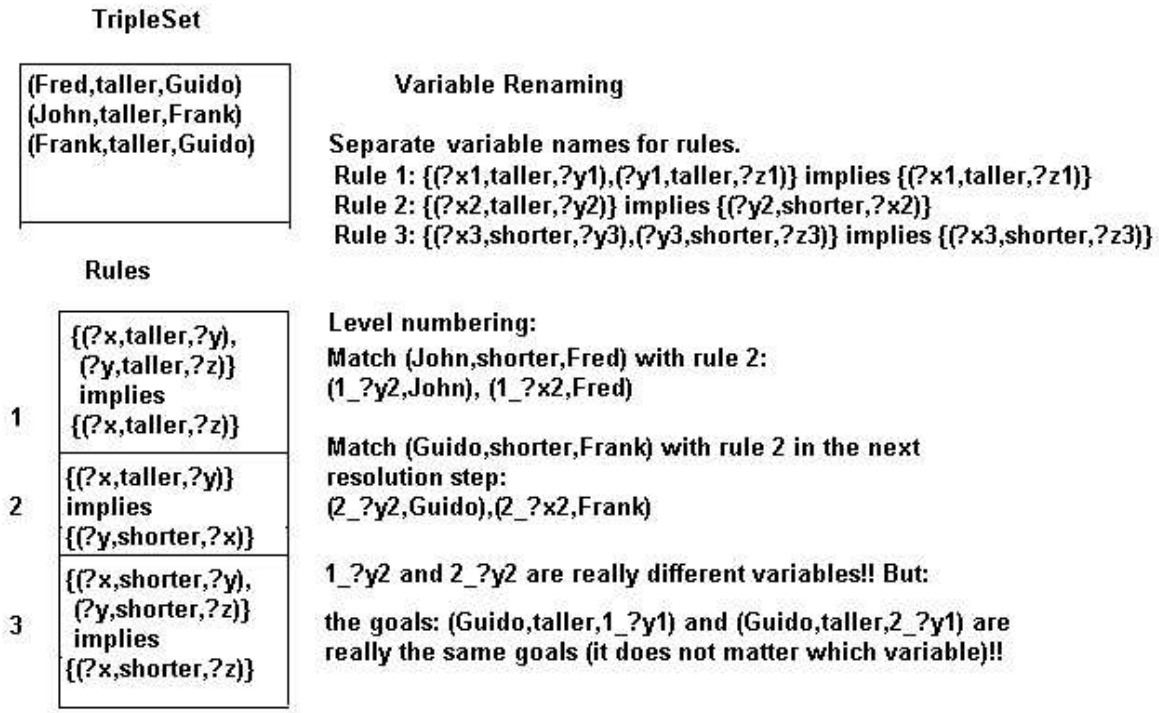

Fig. 5.17. Example of variable renaming.

5.13. Comparison of resolution and closure paths

5.13.1. Introduction

Rules, query and solution have been defined using forward reasoning. However, this thesis is about a backward reasoning engine. In order then to prove the correctness of the followed procedure the connection between backward and forward reasoning must be established.

5.13.2. Elaboration

There is clearly a likeness between closure paths and resolution paths. As well for a closure path as for a resolution path there is a sequence of triples (x_i, r_i, y_i) where x_i and y_i are subgraphs and r_i is a rule. In fact parts of the closure process are done in reverse in the resolution process.
I want to stress here the fact that I define closure paths and resolution paths with the **complete accumulated substitution applied** to all triplesets that are part of the path.
Question: is it possible to find criteria such that steps in the resolution process can be judged to be valid based on (steps in) the closure process?
If a rule is applied in the resolution process to a triple set and one of the triples is replaced by the antecedents then schematically:
(c, as)
where c represent the consequent and as the antecedents.
In the closure process there might be a step:
(as, c) where *c* is generated by the subgraph *as*.

A closure path *generates* a solution when the substituted query matches with a subgraph of the closure produced by the closure path.

I will present some lemmas here with explanations and afterwards I will present a complete theory.

Definitions:
A *final resolution path* is the path that remains after the goallist in the (depth first search) resolution process has been emptied. The path then consists of all the rule applications applied during the resolution process.
A *valid* resolution path is a path where the triples of the query become grounded in the path after substitution or match with the graph *G*; or the resolution process can be further applied to the path so that finally a path is obtained that contains a solution. All resolution paths that are not valid are *invalid* resolution paths. The resolution path is *incomplete* if after substitution the triple sets of the path still contain variables. The resolution path is *complete* when, after substitution, all triple sets in the path are grounded.

Final Path Lemma. The reverse of a final path is a closure path. All final paths are complete and valid.

I will explain the first part here and the second part later.

Take a one step resolution path *(q, r_1, g_1)*. Thus the query matches once with rule r_1 to generate subgraph g_1 who matches with graph *G*. This corresponds to the closure path: *(g_1, r_1, q)* where *q* in the closure *G'* is generated by rule r_1.
Example: Graph *G* :
{(chimp, subClassOf, monkeys),(monkeys, subClassOf, mammalia)}
And the subclassof rule:
{(?c1, subClassOf, ?c2),(?c2, subClassOf, ?c3)} implies {(?c1, subClassOf, ?c3)}
- The closure generates using the rule:
(chimp, subClassOf, mammalia)
and adds this to the database.
- The query:
{(chimp, subClassOf, ?who)}
will generate using the rule :
{(chimp, subClassOf, ?c2),(?c2, subClassOf, ?c3)}
and this matches with the database where *?c2* is substituted by *monkeys* and *?c3* is substituted by *mammalia*.

Proof:
Take the resolution path:
(c_1, r_1, g_1), (c_2, r_2, g_2), … (c_{n-1}, r_{n-1}, g_{n-1}),(c_n, r_n, g_n)
This path corresponds to a sequence of goals in the resolution process:
c_1, $c_{2,\ldots,}$ c_{n-1}, c_n. One moment or another during the process these goals are in the goallist. Goals generate new goals or are taken from the goallist when they are grounded. This means that if the goallist is empty all variables must have been substituted by **URI**'s or literals. But perhaps a temporary variable could exist in the sequence i.e. a variable that appears in the antecedents of a rule *x*, also in the consequents of a rule *y* but that has disappeared in the antecedents of rule *y*. However this means that there is a variable in the consequents of rule *y* that is not present in the antecedents what is not permitted.

Definition: a *minimal closure path* with respect to a query is a closure path with the property that the graph produced by the path matches with the query and that it is not possible to diminish the length of the path i.e. to take a rule away.

Solution Lemma I: A solution corresponds to an empty closure path (when the query matches directly with the database) or to a minimal closure path.

Proof: This follows directly from the definition of a solution as a subgraph of the closure graph. There are two possibilities: the triples of a solution either

belong to the graph *G* or are generated by the closure process. If all triples belong to *G* then the closure path is empty. If not there is a closure path that generates the triples that do not belong to *G*.

It is clear that there can be more than one closure path that generates a solution. One must only consider applying rules in a different sequence. As the reverse of the closure path is a final resolution path it is clear that solutions can be duplicated during the resolution process.

Solution Lemma II: Be *C* the set of all closure paths that generate solutions. Then there exist matching resolution paths that generate the same solutions.

Proof: Be $(g_1, r_1, c_1),(g_2, r_2, c_2), \ldots,(g_n, r_n, c_n)$ a closure path that generates a solution.
Let $(c_n, r_n, g_n), \ldots (c_2, r_2, g_2),(c_1, r_1, g_1)$ be the corresponding resolution path.
The applied rules are called $r_1, r_2, \ldots, r_n$.
r_n applied to c_n gives g_n in the resolution path. Some of the triples of g_n can match with *G*, others will be proved by further inferencing (for instance one of them might match with the tripleset c_{n-1}).When c_n was generated in the closure it was deduced from g_n. The triples of g_n can be part of *G* or they were derived with rules $r_1, r_2, \ldots r_n$. If they were derived then the resolution process will use those rules to inference them in the other direction. If a triple is part of *G* in the resolution process it will be directly matched with a corresponding triple from the graph *G*.

Final Path Lemma. The reverse of a final path is a closure path. All final paths are complete and valid.
I give now the proof of the second part.
Proof: Be $(c_n, r_n, g_n), \ldots (c_2, r_2, g_2),(c_1, r_1, g_1)$ a final resolution path. I already proved in the first part that it is grounded because all closure paths are grounded. g_1 is a subgraph of *G* (if not, the triple (c_1, r_1, g_1) should be followed by another one). c_n is necessarily a triple of the query. The other triples of the query match with the graph *G* or match with triples of c_i, c_j etc.. (consequents of a rule). The reverse is a closure path. However this closure path generates all triples of the query that did not match directly with the graph *G*. Thus it generates a solution.
An example of closure path and resolution path as they were produced by a trace of the engine can be found in appendix 2.
The final path lemma also proves the soundness of the engine because solutions are the substitutions associated with final paths.

5.14. Anti-looping technique

5.14.1. Introduction

The inference engine is *looping* whenever a certain subgoal keeps returning ad infinitum in the resolution process.
There are two reasons why an anti-looping technique is needed:

1) loopings are inherent with rules that cause recursion in the resolution process. An example of such a rule:
   {(?a, subClassOf, ?b), (?b, subClassOf, ?c)} implies {(?a, subClassOf, ?c)}
   Suppose the query *(s, subClassOf, ?c)* matches with the consequent of the rule. Then the goals *(s, subClassOf, ?b)* and *(?b, subclassOf, ?c)* are generated. But *(s, subClassOf, ?b)* matches again with the consequent etc…
   These rules occur quite often especially when working with an ontology. In fact, it is not even possible to seriously test an inference engine for the Semantic Web without an anti-looping mechanism. A lot of testcases use indeed such rules.
2) an inference engine for the Semantic Web will often have to handle rules coming from the most diverse origins. Some of these rules can cause looping of the engine. This means it is not possible to be sure that such rules are not in the database. However, because everything is meant to be done without human intervention, looping is a bad characteristic. It is better that the engine finishes without result. Then eventually e.g. a forward reasoning engine can be tried.

5.14.2. Elaboration

The technique described here stems from an oral communication of De Roo [DEROO]. Given a resolution path: $(c_n, r_n, g_n), \ldots (c_2, r_2, g_2),(c_1, r_1, g_1)$. Suppose the goal c_2 matches with the rule r_2 to generate the goal g_2. When that goal is identical to one of the goals already present in the resolution path and generated also by the same rule then a loop may exist where the goals will keep coming back in the path.
Take the resolution path:
$(c_n, r_n, g_n), \ldots(c_x, r_x, g_x) \ldots(c_y, r_y, g_y)\ldots (c_2, r_2, g_2),(c_1, r_1, g_1)$
where the triples $(c_x, r_x, g_x),(c_y, r_y, g_y)$ and (c_2, r_2, g_2) are equal (recall that the numbers are sequence numbers and not identification numbers).

I call, by definition, a *looping path* a path in which the same triple (c_x, r_x, g_x) occurs twice.
The anti-looping technique consists in failing such a path from the moment a triple (c_x, r_x, g_x) comes back for the second time. These triples can still contain variables so the level numbering has to be stripped of the variables before the comparison.

No solutions are excluded by this technique as shown by the solution lemma II. Indeed the reverse of a looping path is not a closure path. But there exists closure paths whose reverse is a resolution path containing a solution. So all solutions can still be found by finding those paths.

In **RDFE**ngine I control whether the combination antecedents-rule returns which is sufficient because the consequent is determined then.
In the implementation a list of antecedents is kept. This list must keep track of the current resolution level; when backtracking goals with a higher level than the current level must be token away.

That this mechanism stops all looping can best be shown by showing that it limits the maximum depth of the resolution process, however without excluding solutions.
When a goal matches with the consequent of rule *r* producing the antecedents *as*, the tuple *(as, r)* is entered into a list, called the oldgoals list. Whenever this tuple *(as, r)* comes back in the resolution path backtracking is done. This implies that in the oldgoals list no duplicate entries can exist. The number of possible entries in the list is limited to all possible combinations *(as, r)* that are finite whenever the closure graph *G'* is finite. Let this limit be called *maxG*. Then the maximum resolution depth is $maxR = maxG + maxT$ where *maxT* is the number of grounded triples in the query and *maxG* are all possible combinations *(as, r)*.
Indeed, at each step in the resolution process either a rule is used generating a tuple *(as, r)* or a goal(subgraph) is matched with the graph *G*.
When the resolution depth reaches *maxR* all possible combinations *(as, r)* are in the oldgoals list; so no rule can produce anymore a couple *(as, r)* that is not rejected by the anti-looping technique.
In the oldgoals list all triples from the closure graph *G'* will be present because all possible combinations of goals and rules have been tried. This implies also that all solutions will be found before this maximum depth is reached.
Why does this technique not exclude solutions?
Take a closure path $(as_1, r1, c1) \ldots (as_n, r_n, c_n)$ generating a solution . No rule will be applied two times to the same subgraph generating the same consequent to be added to the closure. Thus no as_x is equal to as_y. The reverse of a closure path is a resolution path. This proofs that there exist resolution paths generating a solution without twice generating the same tuple *(as, r)*.
A maximal limit can be calculated for *maxG*.
Be *atot* the total number of antecedents generated by all rules and *ntot* is the total number of labels and variables. Each antecedent is a triple and can exist maximally in $ntot^3$ versions. If there are n_1 grounded triples in the query this gives for the limit of *maxG*: $(ntot^3)^{(atot + n1)}$.
This can easily be a very large number and often stack overflows will occur before this number is reached.

5.14.3. Failure

A failure (i.e. when a goal does not unify with facts or with a rule) does never exclude solutions. Indeed the reverse of a *failure resolution path* can never be a closure path. Indeed the last triple set of the failure path is not included in the closure. If it were its triples should either match with a fact or with the consequent of a rule.

5.14.4. Soundness and completeness

Soundness was demonstrated in 5.13.2.
Each solution found in the resolution process corresponds to a resolution path. Be *S* the set of all those paths. Be *S1* the set of the reverse paths which are valid closure paths. The set *S* will be *complete* when the set *S1* contains all possible closure paths generating the solution. This is the case in a resolution process that uses depth first search as all possible combinations of the goal with the facts and the rules of the database are tried.

5.14.5. Monotonicity

Suppose there is a database, a query and an answer to the query. Then another database is merged with the first. The query is now posed again. What if the first answer is not a part of the second answer?
In this framework monotonicity means that when the query is posed again after the merge of databases, the first answer will be a subset of the second answer. The above definition of databases, rules, solutions and queries imply monotonicity. Indeed all the facts and rules of the first database are still present so that all the same triples will be added during the closure process and the query will match with the same subgraphs.
Nevertheless, some strange results could result.
Suppose a tripleset:
{(blood, color, ”red”)}
and a query:
{(blood, color, ?what)}
gives the answer:
{(blood, color, ”red”)}.
Now the tripleset:
{(blood, color, ”yellow”)}
is added. The same query now gives two answers:
{(blood ,color, ”red”)} and *{(blood, color, ”yellow”)}.*
However this is normal as nobody did ‘tell’ the computer that blood cannot be yellow.

To enforce this, extensions like **OWL** are necessary where a restriction is put on the property ‘color’ in relation with the subject ‘blood’.

In appendix 5 a complete theory is proposed.
Fig. 5.18. gives an overview of the lemmas and the most important points proved by them.

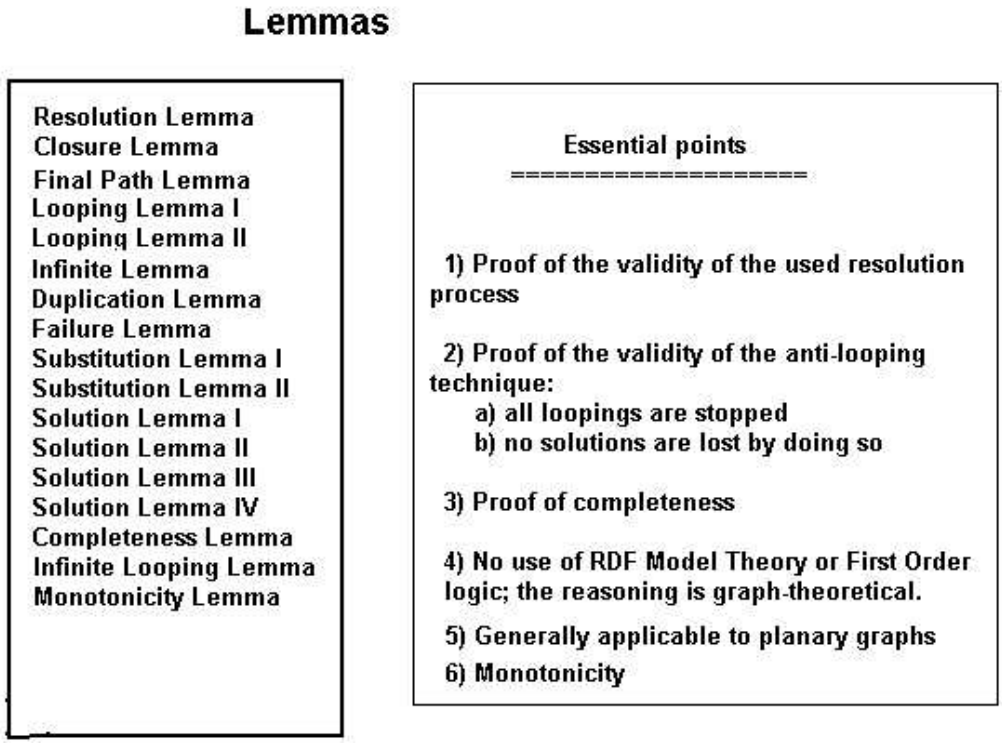

Fig.5.18. Overview of the lemmas.

5.15. The basic IO loop and subquerying

5.15.1. Introduction

The testcase in chapter 1 shows that, during the inferencing process on the main server, queries have to be directed to other servers. For instance, a query will have to be directed to the French railways. I will call this subquerying. In order to enable subquerying, it is necessary that the inference process can be interrupted. This is made possible by the definition of an inference step and the accompanying data structure. The inference steps and the subqueries are executed in the basic **IO** loop.

5.15.2. Definitions

An *inference server* is a server that can accept a query and return a result. Query and result are in a standardized format.
A *RDF server* is a server that can do inferencing on one or more **RDF** graphs. The inferencing process uses a list of graphs. Graphs can be loaded and unloaded. The inferencing is done over all loaded graphs.
In a process of subquerying, there is a *primary server* and one or more *secondary servers*. The primary server is the server where the inferencing process is started. The secondary servers execute part of the inferencing process.
An *inference step* is the inferencing unit in an inferencing environment. In **RDFE**ngine an inference step corresponds to the unification of one goal statement with the array of **RDF** graphs.

5.15.3. Mechanisms

The program **RDFE**ngine is organised around a basic **IO** loop with the following structure:
1) get a command from the console or a file; in automatic mode, this is a null command.
2a) the command is independent from the inferencing process. Execute the command. Goto 1.
2b) read a statement from the goallist. Goto 3.
3) call an **API** or execute a builtin or do an inference step.
4) show or print results, eventually in a trace file. Goto 1.

Fig. 5.19 gives a view of the help function.

The **API** permits to add builtins. Builtins are statements with special predicates that are handled by specialized functions. An example might be a ‘print’ predicate that performs an **IO**.
Example: a fact with the triplelist *[(Txxx,print,”This is a message.”)]*.

A query can be directed to a *secondary server*. This is done by using the predicate *“query”* in a triple *(Txxx,query,”server”)*. This triple is part of the antecedents of a rule. When the rule is selected the antecedents of the rule are added to the goallist. All goals after the goal with predicate “query” and before a goal with predicate *“end_query”* will compose the subquery. This query is sent to the secondary server that returns a failure or a set of solutions. A solution is a the substituted query plus a proof. The primary server can eventually proceed by doing a verification of the proof. Then the returned results are added to the datastructure InfData of the primary server who proceeds with the inference process.

```
Enter command (? for help):
?
h       : help
?       : help
q       : exit
s       : perform a single inference step
g       : go; stop working interactively
m       : menu
e       : execute menu item
qe      : enter a query
```

Fig.5.19. The interactive interface.

This mechanism defines an *interchange format* between inferencing servers. The originating server sends a query, consisting of a tripleset with existential and universal variables. Those variables can be local (scope: one triple) or global (scope: the whole triplelist). The secondary server returns a solution: if not empty, it consists of the query with at least one variable substituted and of a proof. The primary server is not concerned with what happens at the secondary server. The secondary server does not even need to work with **RDF**, as long as it returns a solution in the specified format.
Clearly, this call on a secondary server must be done with care. In a recursive procedure this can easily lead to a kind of spam, thereby overloading the secondary server with messages.
A possibility for reducing the traffic consists in storing the results of the query in a rule with the format: *{query} log:implies {result}*. Then whenever the query can be answered using the rule, it will not be sent anymore to the secondary server. This constitutes a kind of learning process.

This mechanism can also be used to create *virtual servers* on one system, thereby creating a modular system where each *"module"* is composed of several **RDF** graphs. Eventually servers can be added or deleted dynamically.

5.15.4. Conclusion

The basic **IO** loop and the definition of the inferencing process on the level of one inference step makes a highly modular structure possible. It enables also, together with the interchange format, the possiblility to direct queries to other servers by use of the 'query' and 'end_query' predicates.

The use of **ADT**'s and mini-languages makes the program robust because changes in data structures are encapsulated by the mini-languages.
The basic **IO** loop makes the program extensible by giving the possibility of adding builtins and doing **IO**'s in those builtins.

5.16. An evaluation of RDFEngine

5.16.1. Introduction

An inference engine for the Semantic Web should be capable to implement the requirements that are a consequence of its use in the Semantic Web. The degree in which **RDFE**ngine implements some major requirements, is shown.

5.16.2. Elaboration

1) the capability for handling large data sets. This follows from the case study (See 1.2.). It must be possible to consult a large database. **RDFE**ngine retrieves its triples from a hashtable. Instead of a hashtable, the triples can be retrieved from a relational database, using **SQL** statements, with only minor modifications to the program.
2) the engine must be easily extensible. The basic **IO**-loop in **RDFE**ngine (See 5.15.) permits an easy addition of extensions.
3) the engine must be capable of doing subqueries. **RDFE**ngine implements this (See 5.15.). More research is needed concerning the precise modalities of the subquerying.
4) the engine must be capable of working as well with open sets as with closed sets (See 4.5.). **RDFE**ngine only handles open sets. For closed sets, the addition of an extra module is necessary.
5) all triples in a result must be verifiable (2.4.2.). **RDFE**ngine keeps a record of all the namespaces of all labels . The verification itself is not implemented. More research is needed.
6) a trust system should decide about the truth value of triples. **RDFE**ngine makes no connection with a trust system.
7) the engine must be capable of dealing with inconsistencies. In the basic version of **RDFE**ngine, inconsistencies cannot arise (See 7.2.).

5.17. Applications of the engine

5.17.1. Introduction

The Semantic Web is a very new field of research. This makes it necessary to test a lot in order to avoid ending with theoretic constructions that are not usable in daily practice. This makes testcases a very important subject.
I will discuss here the most important testcases. Pieces of source code will be given in **RDFP**rolog syntax. The full original sources are given in appendix 3.

5.17.2. Gedcom

Gedcom is a system that was developed to provide a flexible, uniform format for exchanging computerized genealogical data. The gedcom testcase of De Roo [DEROO] gives a file with rules and file with facts. A query can then be done. This testcase is very good for testing a more complex chaining of rules; also for testing with a large set of data.
Example rule:
gc:parent(Child, Parent),gc:sex(Parent, male) :> gc:father(Child, Parent).
Two facts:
gc:parent(frank, guido),gc:sex(guido, male).
The query:
gc:father(Who, guido)
will give as solution:
gc:father(frank, guido).

5.17.3. The subClassOf testcase

This testcase is an example where a general rule (with a variable predicate) will cause looping if there is no anti-looping mechanism available.
It contains following facts and rules:
a(rdfs:subClassOf, owl:TransitiveProperty)
a(P, owl:TransitiveProperty), P(C1, C2), P(C2, C3) :> P(C1, C3).
Given the facts:
rdfs:subClassOf (mammalia, vertebrae).
rdfs:subClassOf (rodentia, mammalia)
and the query:
rdfs:subClassOf(What, vertebrae).
The result will be:
rdfs:subClassOf(mammalia, vertebrae).
rdfs:subClassOf(rodentia, vertebrae).

5.17.4. An Alpine club

Following example was taken from [CLUB].
Brahma, Vishnu and Siva are members of an Alpine club.

Who is a member of the club is either a skier or a climber or both.
A climber does not like the rain.
A skier likes the snow.
What Brahma likes Vishnu does not like and what Vishnu likes, Brahma does not like.
Brahma likes snow and rain.
Which member of the club is a climber and not a skier?

This case is illustrative because it shows that ‘not’ and ‘or’ can be implemented by making them properties. If this is something handy to do remains to be seen but perhaps it can be done in an automatic way. A possible way of handling this is to put all alternatives in a list.
I give the full source here as this is very illustrative for the logic characteristics of **RDF**.
In **RDFP**rolog it is not necessary to write e.g. *climber(X,X)*; it is possible to write *climber(X)*. However it is written as *climber(X,X)* to demonstrate that the same thing can be done in Notation 3 or Ntriples.

Source:
member(brahma, club).
member(vishnu, club).
member(siva, club).
member(X, club) :> or_skier_climber(X, X).
or_skier_climber(X, X), not_skier(X, X) :> climber(X, X).
or_skier_climber(X, X),not_climber(X, X) :> skier(X, X).
skier(X, X),climber(X, X) :> or_skier_climber(X, X).
skier(X, X),not_climber(X, X) :> or_skier_climber(X, X).
not_skier(X, X),climber(X, X) :> or_skier_climber(X, X).

climber(X, X) :> not_likes(X, rain).
likes(X, rain) :> no_climber(X, X).
skier(X, X) :> likes(X, snow).
not_likes(X, snow) :> no_skier(X, X).

likes(brahma, X) :> not_likes(vishnu, X).
not_likes(vishnu, X) :> likes(brahma, X).
likes(brahma, snow).
likes(brahma, rain).

Query: *member(X, club), climber(X, X),no_skier(X, X).*

Hereafter is the same source but with the modifications to **RDF** as proposed in chapter 4:

member(brahma, club).
member(vishnu, club).
member(siva, club).

member(X, club) :> or(skier, climber)(X,X).
climber(X, X) :> not(likes(X, rain)).
skier(X, X) :> likes(X, snow).
not(likes(X, snow)) :> not(skier(X, X)).
likes(bhrama, X) :> not(likes(vishnu, X)).
not(likes(vishnu, X)) :> not(likes(brahma, X)).
likes(brahma, snow).
likes(brahma, rain).

Query: *member(X, club), climber(X, X), not(skier(X, X)).*

5.17.5. A simulation of a flight reservation

I want to reserve a place in a flight going from Zaventem (Brussels) to Rio de Janeiro. The flight starts on next Tuesday (for simplicity only Tuesday is indicated as date).
Three servers are defined:
myServer contains the files: flight_rules.pro and trust.pro.
airInfo contains the file: air_info.pro.
intAir contains the file: International_air.pro.

Namespaces are given by:

http://thesis/inf, abbreviated *inf*, the engine system space

http://thesis:flight_rules, abbreviated *frul*,the flight rules.

http://thesis/trust, abbreviated *trust*, the trust namespace.

http://flight_ontology/font, abbreviated *font*, the flight ontology space

http://international_air/air, abbreviated *air*, the namespace of the International Air company.

http://air_info/airinfo, abbreviated *ainf*, the namespace of the air information service.
These are non-existent namespaces created for the purpose of this simulation..

The process goes as follows:

1) start the query. The query contains an instruction for loading the flight rules.
2) Load the flight rules (module flight_rules.pro) in G2.
3) Make the query for a flight Zaventem – Rio.
4) The flight rules will direct a query to *air info*. This is done using the predicate *query*.
5) Air info answers: yes, the company is International Air.
6) The flight rules will consult the trust rules: module trust.pro. This is done by directing a query to itself.
7) The flight rules ask whether the company International Air can be trusted.
8) The trust module answers yes, so a query is directed to International Air for commanding a ticket.
9) International Air answers with a confirmation of the reservation.

The reservation info is signalled to the user. The user is informed of the results (query + info + proof).

Chapter 6. Optimization

6.1. Introduction

When an inference engine is fed with a query one or more solutions will be found after a certain search time. There are two compelling reasons for trying to minimize this search time.

1) it is possible that the search must manipulate very big data sets

2) in reasoning processes actions that are part of the logic system are applied repeatedly following paths of reasoning of which some lead to a solution. Many other paths however do not lead to a solution. Each step in a path can generate many other paths which can give a potential explosion of search actions. Recall from 5.14.2. a calculation of the maximum resolution depth in **RDFE**ngine $maxR > (ntot^3)^{(atot + n1)}$ which is exponential.

The minimization of the search time is called the optimization of the engine. Optimization can be viewed from different angles.

1) optimization in forward reasoning is quite different from optimization in backward reasoning (resolution reasoning). I will put the accent on resolution reasoning.

2) the optimization can be considered on the technical level i.e. the performance of the implementation is studied. Examples are: use a compiled version instead of an interpreter, use hash tables instead of sequential access, avoid redundant manipulations etc… Though this point seems less important than the following point (3) I will propose nevertheless two optimizations which are substantial.

3) reduce as much as possible the number of steps in the inferencing process. Here it is not only tried to stop the combinatorial explosion but it is also tried to minimize the number of unification loops.

There are two further important aspects:

a) only one solution is needed e.g. in an authentication process the question is asked whether the person is authenticated or not. In this case certain optimizations are possible that cannot be done in case b.
b) all solutions are needed.

From the theory presented in chapter 5 it is possible to deduce an important optimization technique. It is explained in 6.2.

This chapter mainly presents theoretical results and considerations. A few practical results are presented in 6.4. Further empirical research is needed for obtaining a better view on the usability.

6.2. A result from the research

6.2.1. Introduction

The slogan 'proof as program' suggests that a proof in automated reasoning can be transformed to a program. Based on this idea I developed a mechanism for transforming the proof of a query *predicate(S, O)* where predicate is some predicate, to a general rule that permits to find a solution to the query *predicate(S, O)* in one step.

6.2.2. The theory

I will present the following lemma in **RDFP**rolog notation (See 2.8) for clarity.

Lemma.

Given a query:
predicate(S, O) where predicate is some predicate and a proof *P* found by **RDFE**ngine:
$p_1(s_1, o_1), p_2(s_2, o_2), \ldots, p_n(s_n, o_n),$
$[p_{11}(s_{11}, o_{11}), \ldots., p_{1n}(s_{1n}, o_{1n})] \rightarrow p_{c11}(s_{c11}, o_{c11}),$
….
$[p_{n1}(s_{n1}, o_{n1}), \ldots., p_{nn}(s_{nn}, o_{nn})] \rightarrow predicate(s_{cn1}, o_{cn1}).$
The last triple of the last rule is the solution. No rule contains a variable predicate.

In the facts from the proof (*fcts*) and in the solution (*sol*) replace all **URI**'s that were variables in the original rules by a variable. Define a rule: *fcts* → *sol*. This rule is then a rule that gives in one step a solution for the query *predicate(S, O)*.

Proof.

The proof *P* found by **RDFE**ngine is based on forward reasoning (See 5.9 and 5.13). The solution found by **RDFE**ngine is deduced from the given facts using the given rules. This proves the validity of:
$[p_1(s_1, o_1), p_2(s_2, o_2), \ldots, p_n(s_n, o_n)] \rightarrow predicate(s_{cn1}, o_{cn1})$.
Now it must be demonstrated that the replacement of the **URI**'s by variables is permitted.
Suppose a triple $p_1(s_1, o_1)$ unifies with a rule giving a substitution (o_1, X_1) where X_1 is a variable. In the resolution process this variable will unify with another variable etc... giving a substitution list $(o_1, X_1),(X_1, X_2),\ldots,(X_{n-1}, X_n)$. If the resolution path gives a solution then this list will be finite. Now, in instead of the label o_1 a label o_x is used, this chain of substitutions will not be changed. So this change of o_1 to o_x will also lead to a solution. This shows that o_1 can be replaced by a variable.
The only **URI** that is part of the query is *predicate*. This **URI** will never be replaced because the rules do not contain variable predicates.**QED**.

6.2.3. Example

An example will clarify the technique. The example is in **RDFP**rolog.

1) a query, its solution and its proof.
   a) query:

      grandfather(W1, W2).

   b) solution:

      grandfather(frank, pol).

   c) proof:

      childIn (guido, naudts_huybrechs).
      spouseIn (pol, naudts_huybrechs).
      childIn (frank, naudts_vannoten).
      spouseIn (guido, naudts_vannoten).
      sex (pol, m).

      [childIn (guido, naudts_huybrechs), spouseIn (pol, naudts_huybrechs).]
      → [parent (guido, pol).]
      [childIn (frank, naudts_vannoten), spouseIn (guido, naudts_vannoten).]
      → [parent (frank, guido).]
      [parent (frank, guido), parent (guido, pol).] → [grandparent (frank, pol).]
      [grandparent (frank, pol), sex(pol, m).] → [grandfather (frank, pol).]

2) transformation of 1) to a procedure.

In the facts from the proof (*fcts*) and in the solution (*sol*) replace all **URI**'s that are not part of the query and that were variables in the original rules by a variable. Then define a rule: *fcts* → *sol*.
This gives the rule:
[childIn(GD, NH). spouseIn(PL, NH). childIn(F, NV). spouseIn(GD, NV). sex(PL, m).] → *[grandfather(F, PL).]*

6.2.4. Discussion

This technique, which is illustrated by the example above, can be used in two ways:

1) As a learning technique where a rule is added to the database each time a query is executed.
2) As a preventive measure: for each predicate a query is started of the form: *predicate(X, Y).* For each solution that is obtained, a rule is deduced and added to the database. In this way, all queries made of one triple will only need one rule for obtaining the answer.
This can be done, even if large amounts of rules are generated. Eventually the rules can be stored on disk.

For realising point 2 a certain amount of space is necessary for the storage of the generated rules. Also the time necessary to generate the rules must be considered.
A triple can be stored in 12 bytes if each label is given a number (this gives 4 bytes / label, so a considerable number of different labels is possible). With 8 triples in a rule this is about 100 bytes. Lets take 200 bytes for a rule. 1.000.000 rules then take 200.000.000 bytes i.e. less then 200Mb what is in reach for a modern server.
If there are some 10.000 predicates (already a considerable number) then suppose each query (?s,pred,?o) generates 100 solutions and thus 100 rules before the amount of 1.000.000 rules is crossed.
At a certain moment it will be necessary to put the rules on disk.
So space-time considerations start to be relevant at high numbers of rules, when the access time on disk begins to be significant. Then the access time on disk and the generation time of the rules must be weighed against the avoidance of combinatorial explosion.
Of course there is a long learning time i.e. the time to generate all the possible rules. This can be done in background when free cpu-time is available.
Thus two timing factors are relevant:

- the generating time of the rule
- the access time on disk

The technique can also be used as a mechanism for implementing a system of *learning by example.* Facts for an application are put into triple format. Then the user defines rules like:
[son (wim, christine), son (frank, christine)] → [brother (wim, frank)]
The rule is then generalised to:
[son (W, C), son (F, C)] → [brother (W, F)]
After the user introduced some rules more rules can be generated automatically by executing automatically queries like:
predicate (X, Y).

If the inference mechanism has been proven to be error free and the implementation of the mechanism has been proven to be error free and the input is error free, then the generated rule (= procedure) is also error free.

6.3. Other optimization mechanisms.

A substantial amount of research about optimization has been done in connection with **DATALOG** (see 2.8).

1) forward reasoning in connection with **RDF** means the production of the closure graph. I recall that a solution is a subgraph of the closure graph. If e.g. this subgraph contains three triples and the closure involves the addition of e.g. 10000 triples then the efficiency of the process will not be high. The efficiency of the forward reasoning or the bottom-up approach as it is called in the **DATALOG** world is enhanced considerably by using the magic sets technique [YUREK]. Take the query: *grandfather(X, John)* asking for the grandfather of John. With the magic sets technique only the ancestors of John will be considered. This is done by rewriting the query and replacing it with more complex predicates that restrict the possible matches. According to some [YUREK] this makes it better than top-down or backwards reasoning.
2) technical actions for enhancing efficiency. I will only mention here two actions which are specific for the inference engine I developed.
   2a) as was mentioned before, the fact that a predicate is always a variable or a **URI** but never a complex term, permits a very substantial optimization of the engine. Indeed all clauses(triples) can be indexed by their predicate. Whenever a unification has to be done between a goal and the database, all relevant triples can be looked up with the predicate as an index. Of course the match will always have to be done with clauses that have a variable predicate. When this happens in

the query then the optimization cannot be done. However such a query normally is also a very general query. In the database however variable predicates occur mostly in general rules like the transitivity rule. In that case it should be possible to introduce a typing of the predicate and then add the typing to the index. In this way, when searching for alternative clauses it will not be necessary to consider the whole database but only the set of selected triples. This can further be extended by adding subjects and objects to the index. Eventually a general typing system could be introduced where a type is determined for each triple according to typing rules (that can be expressed in Notation 3 for use by the engine).
When the triples are in a database (in the form of relational rows) the collection of the alternatives can be done 'on the fly' by making an (**SQL**)-query to the database. In this way part of the algorithm can be replaced by queries to a database.

2b) the efficiency of the processing can be enhanced considerably by working with numbers instead of with labels. A table is made of all possible labels where each label receives a number. A triple is then represented by three integer numbers e.g. (12, 73, 9). Manipulation (copy, delete) of a triple becomes faster; comparison is just the comparison of integers which should be faster than comparing strings.

3) from the graph interpretation it can be deduced that resolution paths must be searched that are the reverse of closure paths. This implies that a match must be found with grounded triples i.e. first try to match with the original graph and then with the rules. This can be beneficiary for as well the case (3a) (one solution) as (3b) (more than one solution). In the second case it is possible that a failure happens sooner.
It is possible to order the rules following this principle also: try first the rules that have as much grounded resources in their consequents as possible; rules that contain triples like *(?a, ?p, ?c)* should be used last.
The entries in the goallist can be ordered following this principle also: put first in the goallist the grounded entries, then sort the rest following the number of grounded resources.
This principle can be used also in the construction of rules: in antecedents and consequents put the triples with the most grounded resources first.
Generally, in rules triples with variable predicates should be avoided. These reduce the efficiency because a lot of triples will unify with these.

The magic set transformation can also be used in backwards reasoning. Following Yurek this gives a significant reduction in the number of necessary unifications [YUREK].

An interesting optimization is given by the Andorra principle [JONES]. It is based on selection of a goal from the goallist. For each goal the number of matches with the database is calculated. If there is a goal that does not match there is a failure and the rest of the goals can be forgotten. If all goals match, then for the next goal take the goal with the least number of matches.

Prolog and **RDFE**ngine engine take a depth first search approach. Also possible is a breadth first search. In the case only one solution is needed, in general, breadth first search should be more appropriate certainly if the solution tree is very imbalanced. If all solutions must be found then there is no difference. Looping can easily be stopped in breadth first search while a tree is kept and recurrence of a goal can be easily detected.

Typing can substantially reduce the number of unifications. Take e.g. the rule: *{(?p, a, transitiveProperty),(?a, ?p, ?b),(?b, ?p, ?c)} implies {(?a, ?p, ?c)}*

Without typing any triple will match with the consequent of this rule. However with typing, the variable *?p* will have the type *transitiveProperty* so that only resources that have the type transitiveProperty or a subtype will match with *?p*. See also the remarks about typing at the end in appendix 5.

RDFEngine implements two optimizations:

1) the rules are placed after the facts in the database
2) for the unification, all triples with the same predicate are retrieved from a hashtable.

6.4. A few results

The inferencing was done on a Pentium III machine with a Hugs interpreter under Windows 2000.

Fig. 6.1. summarises the results.

Version 1	Version 2	General Rule
76 seconds	10 seconds	5 seconds

Fig. 6.1. Optimisation results.

The inferencing uses the gedcom (genealogical) example of [DEROO] with a query *gd:grandfather(X, Y).* Version one is the classic implementation modelled after [JONES]. Version two uses a hashtable for collecting the relevant predicates for unification. Under the heading “General Rule” a general rule was calculated

for the query (6.2) and then the query was repeated. The calculation time for the rule is **not** added. The general rule is:
gc:sex(X5, m), gc:spouseIn(X5, X1), gc:childIn(X3, X1), gc:spouseIn(X3, X2), gc:childIn(X4, X2) :> gc:grandfather(X4, X5).

This rule was calculated by version 2 of the program. It can be seen that it is indeed a general rule for determining a grandfather.

6.5. Conclusion

A few practical results are obtained with **RDFE**ngine. More experimenting is however necessary to evaluate the usefulness of these results. Many other optimizations are possible of which some are mentioned in this chapter. This is an area where much research can be done.

Chapter 7. Inconsistencies

7.1. Introduction

Inconsistency is a danger for all databases. However, in a stand-alone database inconsistencies can be controlled to a certain degree. On the Semantic Web controlling the inconsistencies is a lot more difficult. During a query, data originating from diverse servers, whose identity is not known beforehand, may be used. Therefore special mechanisms will be necessary to deal with inconsistent and contradictory data on the Semantic Web.
In dealing with inconsistencies a special role is played by logic and by trust systems.

7.2. Elaboration

A lot of things have already been said in chapter 4 dealing with the logics of the Semantic Web.
Two triplesets are inconsistent when:
1) by *human judgement* they are considered to be inconsistent
2) or, by *computer judgement*, they are considered to be inconsistent. The judgement by a computer is done using *logic*. In the light of the **RDF** graph theory a judgement of inconsistency implies a judgement of *invalidity* of the **RDF** graph.

Consider an object having the property 'color'. An object can have only one color in one application; in another application it can have more than one color. In the first case color can be declared to be an *owl:uniqueProperty.* This property imposes a *restriction* on the **RDF** graph: e.g. the triples *(object, color, red)* and *(object, color, blue)* cannot occur at the same time. If they do the **RDF** graph is considered to be invalid. This is the consequence of an implemented logic restriction on an **RDF** graph (in first order logic notation) where:
T1 = (object, color, color1)
T2 = (object, color, color2)
$\forall$*subject, property, object, T1, T2, G: T1* $\wedge$ *T2* $\wedge$ *not equal(color1, color2)* $\wedge$ *element(G, T1)* $\wedge$ *element(G, T2)* $\rightarrow$ *invalid(G)*
However in another application this restriction does not apply and an object can have two colors. This has the following consequence: inconsistencies are *application dependent* . Inconsistencies are detected by applying logic restrictions to two triplesets. If the restriction applies the **RDF** graph is considered to be invalid. The *declaration* and *detection* of inconsistencies is dependent on the logic that is used in the inferencing system.

When a *closed world* logic is applied, sets have boundaries. This implies that inconsistencies can be detected that do not exist in an *open world.*
An example: there is a set of cubes with different colors but none yellow. Then the statement: ‘cube x is yellow’ is an inconsistency. This is not true in an open world.
RDFEngine does not detect any inconsistencies at all. It can however handle applications that detect inconsistencies. Indeed, as said above, this is only a matter of implementing restrictions e.g. with the rule:
{(cubea, color, yellow)} implies {(this, a, inconsistency)}.
if I have an application that does not permit yellow cubes.

Nevertheless, there might be also inherent logic inconsistencies or contradictions.These however are not possible in basic **RDF**. By adding a negation, as proposed in chapter 5, these contradictions can arise e.g.
in **RDFP**rolog:
color(cube1, yellow) and *not(color(cube1, yellow)).*
It is my belief that a negation should be added to **RDF** in order to create the possibility for detecting inconsistencies.
The more logic is added, the more contradictions of these kind can arise. The implementation of an ontology like **OWL** is an example.

7.3. OWL Lite and inconsistencies

7.3.1. Introduction

When using basic **RDF**, inconsistencies can exist but there is no possible way to detect them. Only when an ontology is added, can inconsistencies arise. One of the goals of an ontology is to impose restrictions to the possible values that the elements in a relation can have. In the case of **RDF** this imposes restrictions on subject and object. The introduction of these restrictions introduces necessarily a kind of *negation*. In a way a restriction is a negation of some values. However together with the negation the whole set of problems connected with negation is reintroduced. This involves also the problems of open and closed world and of constructive and paraconsistent logic.

7.3.2. Demonstration

In what follows some elements of **OWL** are considered from the viewpoint of inconsistencies:

rdfs:domain: imposes a restriction on the set to which a subject of a triple can belong.
rdfs:range: imposes a restriction on the set to which an object of a triple can

belong.
Here it must be checked whether a **URI** is a member to a certain class. This membership must have been declared or it should be possible to infer a triple declaring this membership, at least in a constructive interpretation. It can not be inferred from the fact that the **URI** does not belong to a complementary class.

owl:sameClassAs: *(c1, owl:sameClassAs, c2)* is a declaration that *c1* is the same class as *c2*. It is not directly apparent that this can lead to an inconsistency. An example will show this: if a declaration *(c1, owl:disjointWith,c2)* exist, there is an inconsistency.

owl:disjointWith: this creates an inconsistency if two classes are declared disjoint and the same element is found belonging to both. Other inconsistencies are possible: see example above.

owl:sameIndividualAs and *owl:differentIndividualFrom*:
(c1, owl:sameIndividualAs, c2) and *(c1, owl:differentIndividualFrom, c2)* are evidently contradictory.

owl:minimumCardinality and *owl:maximumCardinality*: these concepts put a restriction on het number of possible values that can occur for a certain property

owl:samePropertyAs, *owl:TransitiveProperty*,*owl:SymmetricProperty*: I do not see immediately how this can cause an inconsistency.

7.3.3. Conclusion

Where in the basic **RDF** implementation of the engine contradictions are only application dependent, from the moment an ontology is implemented, the situation becomes a whole lot more complex. Besides the direct inconsistencies indicated above, the few concepts mentioned can give rise to more inconsistencies when they are interacting one with another. I doubt whether I could give an exhaustive list even when I tried. Certainly, a lot more testing with **OWL** should be necessary but this is out of scope for this thesis.

7.4. Using different sources

7.4.1. Introduction

When rules and facts originating from different sites on the internet are merged together into one database, inconsistencies might arise. Two categories can be distinguished:

1) inconsistencies provoked with intent to do harm.
2) involuntary inconsistencies

7.4.2. Elaboration

Category 1: there are, of course a lot of possibilities, and hackers are inventive. Two possibilities:

a) the hacker puts rules on his site or a pirated site which are bound to give difficulties e.g. inconsistencies might already be inherent in the rules.
b) The namespace of someone else is used i.e. the hacker uses the namespace(s) of somebody else.

Category 2: involuntary errors.

a) syntax errors: these should be detected.
b) logical errors: can produce faulty results or contradictions. But what can be done against a bad rule e.g. a rule saying all mammals are birds? A contradiction will only arise when there is a rule (rules) saying that mammals are not birds or this can be concluded.
   This is precisely one of the goals of an ontology: to add restrictions to a description. These restrictions can be checked. If it is declared that an object has a minimum size and a maximum size and somebody describes it with a size greater than the maximum size the ontology checker should detect this.

When inconsistencies are detected, it is important to trace the origin of the inconsistencies (which is not necessarily determined by the namespace). This means all clauses in the database have to be marked. How can the origin be traced back to an originating site?
In this context namespaces are very important. The namespace to which **RDF** triples belong identify their origin. However a hacker can falsify namespaces. Here again the only way to protect namespaces is by use of cryptographic methods.

When two contradictory solutions are found or two solutions are found and only one is expected, a list of the origins of the clauses used to obtain the solutions can eventually reveal a difference in origin of the clauses. When no solution is found, it is possible to suspect first the least trusted site. How to detect who is responsible?

I belief truth on the Semantic Web is determined by trust. A person believes the sites he trusts. It is therefore fundamental in the Semantic Web that every site can be identified unequivocally. This is far from being so on the present web, leaving the door open for abuses.
Each person or automated program should dispose of a system that ranks the ‘trustworthiness’ of sites, ontologies or persons.

Heflin [HEFLIN] has some ideas on inconsistency. He gives the example: if someone states ‘A’ and another one states ‘not A’, this is a contradiction and all conclusions are possible. However this is not possible in **RDF** as a ‘not’ does not exist. Somebody might however claim: ‘X is black’ and another one ‘X is white’. This however will not exclude solutions; it will not lead to whatever conclusion; it just can add extra solutions to a query. Those solutions might be semantically contradictory for a human interpreter. If this is not a wanted result, then ontologies have to be used. If ontologies produce such a contradiction then, of course, there is an incompatibility problem between ontologies.
As a remedy against inconsistency Heflin cites *nonmonotonicity*. He says: ”Often, an implicit assumption in these theories is that the most recent theory is correct and that it is prior beliefs that must change”. This seems reasonable for a closed world; however on the internet is seems more the contrary that is reasonable. As Heflin says: “if anything, more recent information that conflicts with prior beliefs should be approached with skepticism.”, and:
“The field of belief revision focuses on how to minimize the overall change to a set of beliefs in order to incorporate new inconsistent information.”.
What an agent knows is called his *epistemic state*. This state can change in three ways: by expansion, by revision and by contraction. An expansion adds new non-contradictory facts, a revision changes the content and a retraction retrieves content. But do we, humans, really maintain a consistent set of beliefs? Personally, I don’t think so.
Heflin speaks also about *assumption-based truth maintenance systems* (ATMS) . These maintain multiple contexts where each context represents a set of consistent assumptions.

The advantage of **RDF** is that ‘not’ and ‘or’ cannot be expressed, which ensures the monotonicity when merging **RDF** files. However this is not anymore the case with **OWL**, not even with **OWL** Lite.

7.5. Reactions to inconsistencies

When a query is executed by an inference engine and an inconsistency is detected during the execution, the engine should react to this inconsistency. Its reactions are based on a policy. A policy is in general a set of facts and rules.
Two reactions are possible:

1) the computer reacts completely autonomously based on the rules of the policy.
2) the user is warned. A list of reactions is presented to the user who can make a choice.

7.5.1. Reactions to inconsistency

1) the computer gives two contradictory solutions.
2) the computer consults the trust system for an automatic consult and accepts one solution
3) the computer consults the trust system but the trust system decides what happens
4) the computer rejects everything
5) the computer asks the user

7.5.2. Reactions to a lack of trust

1) the information from the non-trusted site is rejected
2) the user is asked

7.6. Conclusion

Inconsistencies can arise inherently as a consequence of the used logic. These kind of inconsistencies can be handled generically. When devising ontology systems inconsistencies and how to handle them should be considered. In the constructive way developed in this thesis, an inconsistency can lead to two contradictory solutions but has no further influence on the inferencing process. Other inconsistencies can be caused by specific applications. To get a systematic view on this kind of inconsistencies more experience with applications will be necessary.

Chapter 8. Results and conclusions

8.1. Introduction

In this chapter the goals of the thesis that were exposed in chapter one are reviewed and results and conclusions are presented.

8.2. Review of the research questions

The Semantic Web is a vast, new and highly experimental domain of research. This thesis concentrates on a particular aspect of this research domain i.e. **RDF** and inferencing. In the layer model of the Semantic Web given in fig. 3.1. this concerns the logic, proof and trust layers with the accent on the logic layer.
In chapter 1 the research questions and tasks of the thesis were exposed. I will repeat them here and discuss the answers and results found.

1) define an inference process on top of RDF and give a specification of this process in a functional language.

Explanation:

In chapter 5 a specification in Haskell of the necessary data structures for an **RDF** inference engine is given as well as a discussion of the implementation of the inference processes in Haskell. Two main versions of an inference engine have been developed. One was very well tested with many testcases but has a rather classical structure; another version has a structure that is better adapted for use in the Semantic Web but is less well tested.

Results:

- specification of data structures in Haskell
- implementation of an inference engine in Haskell in different versions

Conclusions:

an inference engine for the Semantic Web must be able to handle requirements that are specific for the Semantic Web (5.16.2.).

2) which kind of logic is best suited for the Semantic Web.

Explanation:

In chapter 4 the logic characteristics of **RDF** with an implication added were investigated.

Conclusion:

I recommend the use of constructive logic for use in the Semantic Web. An inference engine for **RDF** must necessarily make a difference between open and closed sets.

3) is the specified inference process sound and complete?

Explanation:

In chapter 5 a resolution theory is presented that is based on reasoning on a graph. The theory establishes the validity of the found solutions by **RDFE**ngine as well as the completeness of the engine.

Results:

graph theory of resolution based inferencing

Conclusion:

the inference process as implemented in **RDFE**ngine gives valid results and is complete.

4) what can be done for augmenting the efficiency of the inference process?

Explanation:

During the course of the research many ways for optimizing **RDFE**ngine were found in the literature or discovered. There was no time for empirical research but some interesting theoretical results are obtained.

Results:

a major optimization technique based on graph theory.

Conclusion:

Many optimizations are possible. Further research is necessary.

5) how can inconsistencies be handled by the inference process?

Explanation:

The use of data originating from different internet sites will undoubtedly generate inconsistencies. However, as also Heflin states, at the present moment the precise way in which this will happen is very difficult to assess at the moment.

Conclusions:

the discussion about inconsistencies is necessarily limited at the moment to theoretical considerations. The inconsistencies that will arise really on the World Wide Web will probably in many cases surprise.

The accent has been on points 1,2 and 3. This is largely due to the fact that I encountered substantial theoretical problems when trying to find an answer for these questions. Certainly, in view of the fact that the Semantic Web must be fully automated, very clear definitions of the semantics are needed in terms of computer operations.

Besides the answers to the research questions a word about the experience with Haskell is in place.
An executable engine has been built in Haskell. The characteristics of Haskell force the programmer to execute a separation of concerns. The higher order functions permit to write very compact code. The following programming model was used:

1) define an abstract syntax corresponding to different concrete syntaxes like **RDFP**rolog and Notation 3.
2) encapsulate the data structures into an Abstract Data Type (**ADT**).
3) define an interface language for the **ADT**'s, also called mini-languages.
4) program the main functional parts of the program using these mini-languages.

This programming model leads to very portable, maintainable and data independent program structures.

8.3. Suggestions for further research

Following suggestions can be made for further research:

- collect empirical data about the different propositions for optimization made in chapter 6.
- **RDFE**ngine can be used as a simulation tool for investigating the characteristics of subquerying for several different vertical application models. Examples are: placing a command, making an appointment with a dentist, searching a book, etc…
- the integration with trust systems and cryptographic methods needs further study.
- an interesting possibility is the build of inference engines that use both forward en backward reasoning or a mix of both.
- complementary to the previous point, the feasability of an inference engine based on subgraph matching technologies using the algorithm of Carroll is worth investigating.
- the applicability of methods for enhancing the efficiency of forwards reasoning in an **RDF** engine should be studied. It is especially worth studying whether **RDF** permits special optimization techniques.

- this is the same as previous item but for backward reasoning.
- the interaction between different inference engines can be further studied.
- **OWL** (Lite) can be implemented and the modalities of different implementations and ways of implementation can be studied.
- which user interfaces are needed? What about messages to the user?
- a study can be made what meta information is interesting to provide concerning the inference clients, engines, servers and other entities.
- a very interesting and, in my opinion, necessary field of research is how Semantic Web engines will have learning capabilities. A main motivation for this is the avoidance of repetitive queries with the same result. Possibly also, clients can learn from the human user in case of conflict.
- adding to the previous point, the study of conflict resolving mechanisms is interesting. Inspiration can be sought with the **SOAR** project.
- further study is necessary concerning inferencing contexts. The Ph.D. thesis of Guha [GUHA] can serve as a starting point.
- how can proofs be transformed to procedures and this in relation to learning as well as optimization?
- implement a learning by example system as indicated in chapter 6.
- the definition of a tri-valued constructive logic (true, false, don’t know).

Bibliography

[AIT] H.Aït-Kaci, *WAM A tutorial reconstruction*, Simon Fraser University, Canada, february 18, 1999.
[ADDRESSING] D.Connolly, *Naming and Addressing: URIs, URLs, ...,* W3C, July 9, 2002, www.w3.org/Addressing/ .
[ALBANY] R.Gilmour, *Metadata in practice,*University of North-Carolina, www.albany.edu/~gilmr/metadata/rdf.ppt
[ALVES] H.F.Alves, *Datalog,programmation logique et négation,* IV Semana de Informática da Universidade Federal de Uberlândia, MG, Brasil, 1997.
[ARTEMOV1] S.N.Artemov,*Explicit provability :the intended semantics for intuistionistic and modal logic,* September 1998.
[ARTEMOV2] S.N.Artemov, *Understanding Constructive Semantics (Spinoza lecture),*August 17,1999.
[BERNERS] T.Berners-Lee e.a., *Semantic Web Tutorial Using N3,* May 20,2003, www.w3.org/2000/10/swap/doc/
[BOLLEN] Bollen e.a., *The World-Wide Web as a Super-Brain: from metaphor to model,* Free University of Brussels, 1996, http://pespmc1.vub.ac.be/papers/WWWSuperBRAIN.html
[BOTHNER] Per Bothner, *What is Xquery?,* XML.com , O'Reilly & Associates, Inc, http://www.xml.com/lpt/a/2002/10/16/xquery.html
[BUNDY] A.Bundy , *Artificial Mathematicians,* May 23, 1996, www.dai.ed.ac.uk/homes/bundy/tmp/new-scientist.ps.gz
[CARROLL] J.Carroll, *Matching RDF Graphs,*Hewlett-Packard Laboratories, Bristol, July 2001, www-uk.hpl.hp.com/people/jjc/tmp/matching.pdf.
[CASTELLO] R.Castello e.a. *Theorem provers survey.* University of Texas at Dallas, July 2001, http://citeseer.nj.nec.com/409959.html
[CENG] A.Schoorl, *Ceng 420 Artificial intelligence* University of Victoria, 1999, www.engr.uvic.ca/~aschoorl/ceng420/
[CHAMPIN] P.A.Champin, April 5,2001, http://www710.univ-lyon1.fr/~champin/rdf-tutorial/node12.html
[CLUB] I.Benche, *CIS 571 Unit 3,*University of Oregon, www.cirl.uoregon.edu/~iustin/cis571/unit3.html
[COQ] G.Huet e.a., *RT-0204-The Coq Proof Assistant : A Tutorial,*INRIA, August 1997, www.inria.fr/rrrt/rt-0204.html
[CS] D.Bridge, *Resolution Strategies and Horn Clauses,* Department of Computer Science, University College Cork, Ireland, www.cs.ucc.ie/~dgb/courses/pil/ws11.ps.gz
[CWM] S.B.Palmer, *CWM – Closed World Machine,* May 7, 2001, http://infomesh.net/2001/cwm
[DAML+OIL] D.Connolly e.a., DAML+OIL (March 2001) Reference Description, W3C Note, December 18, 2001. Latest version is available at www.w3.org/TR/daml+oil-reference.
[DECKER] S.Decker e.a., *A query and Inference Service for RDF,*University of KarlsRuhe, 1998, www.w3.org/TandS/QL/QL98/pp/queryservice.html
[DESIGN] T.Berners-Lee, *Tim Berners-Lee's site with his design-issues articles,* http://www.w3.org/DesignIssues/
[DEROO] J.De Roo, *the site of the Euler program,* www.agfa.com/w3c/jdroo
[DICK] A.J.J.Dick, *Automated equational reasoning and the knuth-bendix algorithm: an informal introduction,* Rutherford Appleton Laboratory Chilton, Didcot OXON OX11 OQ, www.site.uottawa.ca/~luigi/csi5109/church_rosser.doc

[EDMONDS] E.Bruce, *What if all truth is context-dependent?*, Manchester Metropolitan University, February 12, 2001, www.cpm.mmu.ac.uk/~bruce
[GANDALF] *Gandalf Home Page*, http://www.cs.chalmers.se/~tammet/gandalf
[GENESERETH] M.Genesereth, *Course Computational logic*, Computer Science Department, Stanford University, 2003
[GHEZZI] Ghezzi e.a., *Fundamentals of Software Engineering*, Prentice-Hall, 1991
[GRAPH] J.De Roo, *Euler site*, www.agfa.com/w3c/euler/graph.axiom.n3
[GROSOF] B.Grosof e.a., *Introduction to RuleML*,October 29,2002, http://ebusiness.mit.edu/bgrosof/paps/talk-ruleml-jc-ovw-102902-main.pdf
[GUHA] R. V. Guha, *Contexts: A formalization and some applications*, MCC Technical Report No. ACT-CYC-423-91, November 1991
[GUPTA] Amit Gupta & Ashutosh Agte, *Untyped lambda calculus, alpha-, beta- and eta-reductions*, April 28/May 1, 2000, www/cis/ksu.edu/~stefan/Teaching/CIS705/Reports/GuptaAgte-2.pdf
[HARRISON] J.Harrison, *Introduction to functional programming*, 1997
[HAWKE] S.Hawke, *RDF Rules mailing list*, September 11, 2001, http://lists.w3.org/Archives/Public/www-rdf-rules/2001Sep/0029.html
[HEFLIN] J.D.Heflin, *Towards the semantic web: knowledge representation in a dynamic,distributed environment*, Dissertation 2001
[HILOG] D.S.Warren, *Programming in Tabled Prolog*, Department of Computer Science, Stony Brook, July 31, 1999, www.cs.sunysb.edu/~warren/xsbbook/node45.html
[JEURING] J.Jeuring e.a., *Grammars and parsing*, Utrecht University. October-December 2002, www.cs.uu.nl/docs/vakken/gont/
[JEURING1] Jeuring e.a. ,*Polytypic unification*, J.Functional programming, september 1998.
[KERBER] Manfred Kerber, *Mechanised Reasoning*, Midlands Graduate School in Theoretical Computer Science, The University of Birmingham, November/December 1999, www.cs.bham.ac.uk/~mmk/courses/MGS/index.html
[KLYNE]G.Klyne, *Nine by nine:Semantic Web Inference using Haskell*, May, 2003, www.ninebynine.org/Software/swish-0.1.html
[KLYNE2002] G.Klyne,*Circumstance, provenance and partial knowledge*, Maart 13, 2002, www.ninebynine.org/RDFNotes/UsingContextsWithRDF.html
[LAMBDA] D.Miller, *lambda prolog home page*, Last updated: 27 April 2002, www.cse.psu.edu/~dale/lProlog/
[LEHMAN] J.Lehman e.a., *A gentle Introduction to Soar, an Architecture for Human Cognition*, 1990
[LINDHOLM] T.Lindholm, *Exercise Assignment Theorem prover for propositional modal logics*,Helsinki Institute for Information Technology, www.cs.hut.fi/~ctl/promod.ps
[MCGUINESS] D.McGuiness, *Explaining reasoning in description logics*, Ph.D.Thesis , 1999
[MCGUINESS2003] D.McGuiness e.a., *Inference Web:Portable explanations for the web*,Knowledge Systems Laboratory, Stanford University, January, 2003
[MYERS] A.Meyers, *CS611 LECTURE 14 The Curry-Howard Isomorphism*, Cornell University, November 26, 2001, www.cs.cornell.edu/courses/cs611/2001fa/scribe/lecture36.pdf
[OTTER] Otter Home Page, Argonne National Laboratory, September 8, 2003, www.mcs.anl.gov/AR/otter/
[OWL Features] D.McGuinness e.a., *Feature Synopsis for OWL Lite and OWL*,Working Draft 29 July 2002. Latest version is available at www.w3.org/TR/owl-features/
[OWL Issues] M.K.Smith, ed.,*Web Ontology Issue Status*, July 10, 2002, www.w3.org/2001/sw/WebOnt/webont-issues.html

[OWL Reference] D.Connolly e.a.,*OWL Web Ontology Language 1.0 Reference*, W3C Working Draft 29 July 2002. Latest version is available at www.w3.org/TR/owl-ref/
[PFENNING_1999] F.Pfenning e.a., *Twelf a Meta-Logical framework for deductive Systems*, Department of Computer Science, Carnegie Mellon University, 1999
[PFENNING_LF] F.Pfenning, *Home page for logical frameworks*, www-2.cs.cmu.edu/afs/cs/user/fp/www/lfs.html
[RDFM] P.Hayes, ed., *RDF Model Theory*,September 5, 2003, www.w3.org/TR/rdf-mt/
[RDFMS] O.Lassila e.a., *Resource Description Framework (RDF) Model and Syntax Specification*, February 22, 1999, www.w3.org/TR/1999/REC-rdf-syntax-19990222
[RDFPRIMER] B.McBride, ed., *RDF Primer*, W3C Working Draft, September 5, 2003, www.w3.org/TR/rdf-primer/
[RDFRULES] *RDF Rules mail archives*, http://lists.w3.org/Archives/Public/www-rdf-rules/
[RDFSC] D.Brickley e.a., eds, *Resource Description Framework (RDF) Schema Specification 1.0*, W3C Candidate Recommendation, March 27, 2000, www.w3.org/TR/2000/CR-rdf-schema-20000327
[RQL] J.Broekstra, *Sesame RQL: a Tutorial*, May 15, 2002, http://sesame.aidministrator.nl/publications/rql-tutorial.html
[SINTEK] M.Sintek e.a.,*TRIPLE – A query, Inference, and Transformation Language for the Semantic Web*, Stanford University, June 2002, http://triple.semanticweb.org/
[STANFORD] P.Frederic, *"Automated Reasoning"*, The Stanford Encyclopedia of Philosophy (Fall 2001 Edition), Edward N. Zalta (ed.), July 18, 2001, http://plato.stanford.edu/archives/fall2001/entries/reasoning-automated/
[STANFORDP] G.Priest e.a., *"Paraconsistent Logic"*, The Stanford Encyclopedia of Philosophy (Winter 2000 Edition), Edward N. Zalta (ed.), September 24, 1996, http://plato.stanford.edu/archives/win2000/entries/logic-paraconsistent/
[SWAP] T. Berners-Lee, *The site of the CWM inference engine*,June 6, 2003, www.w3.org/2000/10/swap
[SWAPLOG] T.Berners-Lee, http://www.w3.org/200/10/swap/log#
[TBL] T. Berners-Lee e.a., *Weaving the web*, Harper San Francisco, September 22, 1999
[TBL01] T.Berners-Lee e.a., *The semantic web*, Scientific American, May 2001
[TWELF] Pfenning e.a., *Twelf Home Page*, www.cs.cmu.edu/~twelf
[UNICODE] *The unicode standard, a technical introduction*,April 23, 2003, www.unicode.org/unicode/standard/principles.html
[UMBC] T.W.Finin, *Inference in first order logic*,Computer Science and Electrical Engineering, University of Maryland Baltimore Country, www.cs.umbc.edu/471/lectures/9/sld040.htm
[URI] T.Berners-Lee e.a.,*Uniform Resource Identifiers (URI): Generic Syntax*, August, 1998, www.ietf.org/rfc/rfc2396.txt
[USHOLD] M.Uschold, *Where is the semantics of the web?*, The Boeing company, April 10, 2002, http://lsdis.cs.uga.edu/events/Uschold-talk.htm
[VAN BENTHEM] Van Benthem e.a., *Logica voor informatici*, Addison Wesley 1991
[WALSH] T.Walsh, *A divergence critic for inductive proof*, 1/96, Journal of Artificial Intelligence Research, April 29, 1996, www-2.cs.cmu.edu/afs/cs/project/jair/pub/volume4/walsh96a-html/section3_3.html
[WESTER] Wester e.a., *Concepten van programmeertalen*,Open Universiteit Eerste druk 1994
[WOS] Wos e.a., *Automated reasoning*,Prentice Hall, 1984
[YUREK] Yurek , *Datalog bottom-up is the trend in the deductive database evaluation strategy*, University of Maryland, 2002.

List of abbreviations

ADT	:	Abstract Data Type
ALF	:	Algebraic Logic Functional Programming Language
API	:	Application Programming Interface
BHK	:	Brouwer, Heyting, Kolmogorov
CWM	:	Closed World Machine
DTD	:	Document Type Definition
ECQ	:	Ex Contradictione Quodlibet
FOL	:	First Order Logic
FSM	:	Finite State Machine
GUI	:	Graphical User Interface
HTML	:	Hypertext Markup Language
KB	:	Knowledge Base
N3	:	Notation 3
OWL	:	Ontology Web Language
RDF	:	Resource Description Framework
RDFS	:	RDF Schema
SGML	:	Standard General Markup Language
SLD	:	Selection, Linear, Definite
SweLL	:	Semantic Web Logic Language
SQL	:	Structured Query Language
URI	:	Uniform Resource Indicator
URL	:	Uniform Resource Locator
URN	:	Uniform Resource Name
W3C	:	World Wide Web Consortium
WAM	:	Warren Abstract Machine
XML	:	Extensible Markup Language .

Appendix 1. Executing the engine

There are actually two versions of the engine. One, N3Engine has been tested with a lot of test cases but has a rather conventional structure. The other, **RDFE**ngine, is less tested but has a structure that is more directed towards the Semantic Web. It is the version of **RDFE**ngine that has been exposed in the text and of which the source code is added at the end.

Usage of the inference engine N3Engine.
First unzip the file EngZip.zip into a directory.

Following steps construct a demo (with the Hugs distribution under Windows):

1) winhugs.exe
2) :l **RDFP**rolog.hs
3) menu
4) choose one of the examples e.g. en2. These are testcases with the source in Notation 3
5) menuR
6) choose one of the examples e.g. pr12. Those with number 1 at the end give the translation from **RDFP**rolog to Notation3; those with number 2 at the end execute the program.

Usage of the inference engine **RDFE**ngine.
First unzip the file **RDFZ**ip.zip into a directory.

Following steps construct a demo (with the Hugs distribution under Windows):

1) winhugs.exe
2) :l Rpro1.hs
3) pro11
4) repeat the s(tep) instruction

or after loading:

3) start
4) m
5) choose an item

Appendix 2. Example of a closure path.

An example of a resolution and a closure path made with the gedcom example (files gedcom.facts.n3, gedcom.relations.n3, gedcom.query.n3)

First the trace is given with the global substitutions and all calls and alternatives:

Substitutions:

{2$_$17_:child = :Frank. _1?who1 = 2$_$17_:grandparent. 3$_$16_:child = :Frank. 2$_$17_:grandparent = 3$_$16_:grandparent. 4$_$13_:child = :Frank. 3$_$16_:parent = 4$_$13_:parent. 4$_$13_:family = :Naudts_VanNoten. 4$_$13_:parent = :Guido. 7$_$13_:child = :Guido. 3$_$16_:grandparent = 7$_$13_:parent. 7$_$13_:family = :Naudts_Huybrechs. 7$_$13_:parent = :Pol. }.
N3Trace:

[:call {:Frank gc:grandfather _1?who1. }].
:alternatives = {{2$_$17_:child gc:grandparent 2$_$17_:grandparent. 2$_$17_:grandparent gc:sex :M. }
}

[:call {:Frank gc:grandparent 2$_$17_:grandparent. }].
:alternatives = {{3$_$16_:child gc:parent 3$_$16_:parent. 3$_$16_:parent gc:parent 3$_$16_:grandparent. }
}

[:call {:Frank gc:parent 3$_$16_:parent. }].
:alternatives = {{4$_$13_:child gc:childIn 4$_$13_:family. 4$_$13_:parent gc:spouseIn 4$_$13_:family. }
}

[:call {:Frank gc:childIn 4$_$13_:family. }].
:alternatives = {{:Frank gc:childIn :Naudts_VanNoten. }
}

[:call {4$_$13_:parent gc:spouseIn :Naudts_VanNoten. }].
:alternatives = {{:Guido gc:spouseIn :Naudts_VanNoten. }
{:Christine gc:spouseIn :Naudts_VanNoten. }
}

[:call {:Guido gc:parent 3$_$16_:grandparent. }].

:alternatives = {{7$_$13_:child gc:childIn 7$_$13_:family. 7$_$13_:parent gc:spouseIn 7$_$13_:family. }
}

[:call {:Guido gc:childIn 7$_$13_:family. }].
:alternatives = {{:Guido gc:childIn :Naudts_Huybrechs. }
}

[:call {7$_$13_:parent gc:spouseIn :Naudts_Huybrechs. }].
:alternatives = {{:Martha gc:spouseIn :Naudts_Huybrechs. }
{:Pol gc:spouseIn :Naudts_Huybrechs. }
}

[:call {:Pol gc:sex :M. }].
:alternatives = {{:Pol gc:sex :M. }
}
Query:

{:Frank gc:grandfather _1?who1. }

Solution:

{:Frank gc:grandfather :Pol. }

The calls and alternatives with the global substitution applied:

It can be immediately verified that the resolution path is complete.
(The alternatives that lead to failure are deleted).

[:call {:Frank gc:grandfather :Pol.}].
:alternatives = {{:Frank gc:grandparent :Pol. :Pol gc:sex :M. }
}

[:call {:Frank gc:grandparent :Pol. }].
:alternatives = {{:Frank gc:parent :Guido. :Guido gc:parent :Pol. }
}

[:call {:Frank gc:parent :Guido. }].
:alternatives = {{:Frank gc:childIn :Naudts_Vannoten. :Guido gc:spouseIn :Naudts_Vannoten. }
}

[:call {:Frank gc:childIn :Naudts_Vannoten. }].
:alternatives = {{:Frank gc:childIn :Naudts_VanNoten. }
}

[:call {:Guido gc:spouseIn :Naudts_VanNoten. }].
:alternatives = {{:Guido gc:spouseIn :Naudts_VanNoten. }
}

[:call {:Guido gc:parent :Pol. }].
:alternatives = {{:Guido gc:childIn :Naudts_Huybrechs. :Pol gc:spouseIn
:Naudts_Huybrechs. }
}

[:call {:Guido gc:childIn :Naudts_Huybrechs. }].
:alternatives = {{:Guido gc:childIn :Naudts_Huybrechs. }
}

[:call {:Pol gc:spouseIn :Naudts_Huybrechs. }].
:alternatives = {{:Pol gc:spouseIn :Naudts_Huybrechs. }
}

[:call {:Pol gc:sex :M. }].
:alternatives = {{:Pol gc:sex :M. }
}

Query:

{:Frank gc:grandfather _1?who1. }

Solution:

{:Frank gc:grandfather :Pol. }

It is now possible to establish the closure path.

({{:Guido gc:childIn :Naudts_Huybrechs. :Pol gc:spouseIn :Naudts_Huybrechs.
}, {:Guido gc:parent :Pol. })
({{:Frank gc:childIn :Naudts_Vannoten. :Guido gc:spouseIn
:Naudts_Vannoten. }, {:Frank gc:parent :Guido. })

({{:Frank gc:parent :Guido. :Guido gc:parent :Pol. },
{:Frank gc:grandparent :Pol. })

({{:Frank gc:grandparent :Pol. :Pol gc:sex :M. },
{:Frank gc:grandfather :Pol.})

The triples that were added during the closure are:
{:Guido gc:parent :Pol. }
{:Frank gc:parent :Guido. }
{:Frank gc:grandparent :Pol. }
{:Frank gc:grandfather :Pol.}

It is verified that the reverse of the resolution path is indeed a closure path.

Verification of the lemmas

Closure Lemma: certainly valid for this closure path.
Query Lemma: the last triple in the closure path is the query.
Final Path Lemma: correct for this resolution path.
Looping Lemma: not applicable
Duplication Lemma: not applicable.
Failure Lemma: not contradicted.
Solution Lemma I: correct for this testcase.
Solution Lemma II: correct for this testcase.
Solution Lemma III: not contradicted.
Completeness Lemma: not contradicted.

Appendix 3. Test cases

Gedcom

The file with the relations (rules):

```
# $Id: gedcom-relations.n3,v 1.3 2001/11/26 08:42:34 amdus Exp $

# Original http://www.daml.org/2001/01/gedcom/gedcom-relations.xml by Mike Dean (BBN Technologies / Verizon)
# Rewritten in N3 (Jos De Roo AGFA)
# state that the rules are true (made by Tim Berners-Lee W3C)
# see also http://www.w3.org/2000/10/swap/Examples.html

@prefix log: <http://www.w3.org/2000/10/swap/log#> .
@prefix ont: <http://www.daml.org/2001/03/daml+oil#> .
@prefix gc: <http://www.daml.org/2001/01/gedcom/gedcom#> .
@prefix : <gedcom#> .

{{:child gc:childIn :family. :parent gc:spouseIn :family} log:implies {:child gc:parent :parent}} a log:Truth; log:forAll :child, :family, :parent.

{{:child gc:parent :parent. :parent gc:sex :M} log:implies {:child gc:father :parent}} a log:Truth; log:forAll :child, :parent.

{{:child gc:parent :parent. :parent gc:sex :F} log:implies {:child gc:mother :parent}} a log:Truth; log:forAll :child, :parent.

{{:child gc:parent :parent} log:implies {:parent gc:child :child}} a log:Truth; log:forAll :child, :parent.

{{:child gc:parent :parent. :child gc:sex :M} log:implies {:parent gc:son :child}} a log:Truth; log:forAll :child, :parent.

{{:child gc:parent :parent. :child gc:sex :F} log:implies {:parent gc:daughter :child}} a log:Truth; log:forAll :child, :parent.

{{:child gc:parent :parent. :parent gc:parent :grandparent} log:implies {:child gc:grandparent :grandparent}} a log:Truth; log:forAll :child, :parent, :grandparent.
```

```
{{:child gc:grandparent :grandparent. :grandparent gc:sex :M} log:implies
{:child gc:grandfather :grandparent}} a log:Truth; log:forAll :child, :parent,
:grandparent.

{{:child gc:grandparent :grandparent. :grandparent gc:sex :F} log:implies
{:child gc:grandmother :grandparent}} a log:Truth; log:forAll :child, :parent,
:grandparent.

{{:child gc:grandparent :grandparent} log:implies {:grandparent gc:grandchild
:child}} a log:Truth; log:forAll :child, :grandparent.

{{:child gc:grandparent :grandparent. :child gc:sex :M} log:implies
{:grandparent gc:grandson :child}} a log:Truth; log:forAll :child, :grandparent.

{{:child gc:grandparent :grandparent. :child gc:sex :F} log:implies
{:grandparent gc:granddaughter :child}} a log:Truth; log:forAll :child,
:grandparent.

{{:child1 gc:childIn :family. :child2 gc:childIn :family. :child1
ont:differentIndividualFrom :child2} log:implies {:child1 gc:sibling :child2}} a
log:Truth; log:forAll :child1, :child2, :family.

{{:child2 gc:sibling :child1} log:implies {:child1 gc:sibling :child2}} a
log:Truth; log:forAll :child1, :child2.

{{:child gc:sibling :sibling. :sibling gc:sex :M} log:implies {:child gc:brother
:sibling}} a log:Truth; log:forAll :child, :sibling.

{{:child gc:sibling :sibling. :sibling gc:sex :F} log:implies {:child gc:sister
:sibling}} a log:Truth; log:forAll :child, :sibling.

{?a gc:sex :F. ?b gc:sex :M.} log:implies {?a ont:differentIndividualFrom ?b.}.

{?b gc:sex :F. ?a gc:sex :M.} log:implies {?a ont:differentIndividualFrom ?b.}.

{{:spouse1 gc:spouseIn :family. :spouse2 gc:spouseIn :family. :spouse1
ont:differentIndividualFrom :spouse2} log:implies {:spouse1 gc:spouse
:spouse2}} a log:Truth; log:forAll :spouse1, :spouse2, :family.

{{:spouse2 gc:spouse :spouse1} log:implies {:spouse1 gc:spouse :spouse2}} a
log:Truth; log:forAll :spouse1, :spouse2.
```

```
{{:spouse gc:spouse :husband. :husband gc:sex :M} log:implies {:spouse
gc:husband :husband}} a log:Truth; log:forAll :spouse, :husband.

{{:spouse gc:spouse :wife. :wife gc:sex :F} log:implies {:spouse gc:wife :wife}}
a log:Truth; log:forAll :spouse, :wife.

{{:child gc:parent :parent. :parent gc:brother :uncle} log:implies {:child
gc:uncle :uncle}} a log:Truth; log:forAll :child, :uncle, :parent.

{{:child gc:aunt :aunt. :aunt gc:spouse :uncle} log:implies {:child gc:uncle
:uncle}} a log:Truth; log:forAll :child, :uncle, :aunt.

{{:child gc:parent :parent. :parent gc:sister :aunt} log:implies {:child gc:aunt
:aunt}} a log:Truth; log:forAll :child, :aunt, :parent.

{{:child gc:uncle :uncle. :uncle gc:spouse :aunt} log:implies {:child gc:aunt
:aunt}} a log:Truth; log:forAll :child, :uncle, :aunt.

{{:parent gc:daughter :child. :parent gc:sibling :sibling} log:implies {:sibling
gc:niece :child}} a log:Truth; log:forAll :sibling, :child, :parent.

{{:parent gc:son :child. :parent gc:sibling :sibling} log:implies {:sibling
gc:nephew :child}} a log:Truth; log:forAll :sibling, :child, :parent.

{{:cousin1 gc:parent :sibling1. :cousin2 gc:parent :sibling2. :sibling1 gc:sibling
:sibling2} log:implies {:cousin1 gc:firstCousin :cousin2}} a log:Truth;
log:forAll :cousin1, :cousin2, :sibling1, :sibling2.

{{:child gc:parent :parent} log:implies {:child gc:ancestor :parent}} a log:Truth;
log:forAll :child, :parent.

{{:child gc:parent :parent. :parent gc:ancestor :ancestor} log:implies {:child
gc:ancestor :ancestor}} a log:Truth; log:forAll :child, :parent, :ancestor.

{{:child gc:ancestor :ancestor} log:implies {:ancestor gc:descendent :child}} a
log:Truth; log:forAll :child, :ancestor.

{{:sibling1 gc:sibling :sibling2. :sibling1 gc:descendent :descendent1. :sibling2
gc:descendent :descendent2} log:implies {:descendent1 gc:cousin
:descendent2}} a log:Truth; log:forAll :descendent1, :descendent2, :sibling1,
:sibling2.
```

The file with the facts:

```
# $Id: gedcom-facts.n3,v 1.3 2001/11/26 08:42:33 amdus Exp $

@prefix gc: <http://www.daml.org/2001/01/gedcom/gedcom#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@prefix : <gedcom#>.

:Goedele gc:childIn :dt; gc:sex :F.
:Veerle gc:childIn :dt; gc:sex :F.
:Nele gc:childIn :dt; gc:sex :F.
:Karel gc:childIn :dt; gc:sex :M.
:Maaike gc:spouseIn :dt; gc:sex :F.
:Jos gc:spouseIn :dt; gc:sex :M.

:Jos gc:childIn :dp; gc:sex :M.
:Rita gc:childIn :dp; gc:sex :F.
:Geert gc:childIn :dp; gc:sex :M.
:Caroline gc:childIn :dp; gc:sex :F.
:Dirk gc:childIn :dp; gc:sex :M.
:Greta gc:childIn :dp; gc:sex :F.
:Maria gc:spouseIn :dp; gc:sex :F.
:Frans gc:spouseIn :dp; gc:sex :M.

:Ann gc:childIn :gd; gc:sex :F.
:Bart gc:childIn :gd; gc:sex :M.
:Rita gc:spouseIn :gd; gc:sex :F.
:Paul gc:spouseIn :gd; gc:sex :M.

:Ann_Sophie gc:childIn :dv; gc:sex :F.
:Valerie gc:childIn :dv; gc:sex :F.
:Stephanie gc:childIn :dv; gc:sex :F.
:Louise gc:childIn :dv; gc:sex :F.
:Christine gc:spouseIn :dv; gc:sex :F.
:Geert gc:spouseIn :dv; gc:sex :M.

:Bieke gc:childIn :cd; gc:sex :F.
:Tineke gc:childIn :cd; gc:sex :F.
:Caroline gc:spouseIn :cd; gc:sex :F.
:Hendrik gc:spouseIn :cd; gc:sex :M.

:Frederik gc:childIn :dc; gc:sex :M.
:Stefanie gc:childIn :dc; gc:sex :F.
:Stijn gc:childIn :dc; gc:sex :M.
:Karolien gc:spouseIn :dc; gc:sex :F.
```

```
:Dirk gc:spouseIn :dc; gc:sex :M.

:Lien gc:childIn :sd; gc:sex :F.
:Tom gc:childIn :sd; gc:sex :M.
:Greta gc:spouseIn :sd; gc:sex :F.
:Marc gc:spouseIn :sd; gc:sex :M.
```

The queries:

```
# $Id: gedcom-query.n3,v 1.5 2001/12/01 01:21:27 amdus Exp $
# PxButton | test | jview Euler gedcom-relations.n3 gedcom-facts.n3 gedcom-
query.n3 |

@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@prefix gc: <http://www.daml.org/2001/01/gedcom/gedcom#>.
@prefix : <gedcom#>.

# _:x gc:parent :Maaike, :Jos. # ok
# :Maaike gc:son _:x.          # ok
# :Jos gc:daughter _:x.        # ok
# _:x gc:grandfather :Frans.   # ok
# :Maria gc:grandson _:x.      # ok
 :Frans gc:granddaughter _:x. # ok
# :Nele gc:sibling _:x.        # ok
# _:x gc:brother :Geert.       # ok
# _:x gc:spouse :Frans.        # ok
# _:x gc:husband _:y.          # ok
# :Karel gc:aunt _:x.          # ok
# :Nele gc:aunt _:x.           # ok
# :Rita gc:niece _:x.
# :Nele gc:firstCousin _:x.
# :Karel gc:ancestor _:x.      # ok
# :Maria gc:descendent _:x.    # ok
# :Karel gc:cousin _:x.        # ok
```

Ontology

The axiom file:

```
# File ontology1.axiom.n3
# definition of rdfs ontology
# modelled after: A Logical Interpretation of RDF
# Wolfram Conen, Reinhold Klapsing, XONAR IT and Information Systems
# and Software Techniques Velbert, Germany U Essen, D-45117

@prefix log: <http://www.w3.org/2000/10/swap/log#>.
@prefix ont: <http://www.w3.org/2002/07/daml+oil#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <ontology#>.

:mammalia rdfs:subClassOf :vertebrae.
:rodentia rdfs:subClassOf :mammalia.
:muis rdfs:subClassOf :rodentia.
:spitsmuis rdfs:subClassOf :muis.

rdfs:subClassOf a owl:TransitiveProperty.
 {{:p a owl:TransitiveProperty. :c1 :p :c2. :c2 :p :c3} log:implies {:c1
 :p :c3}} a log:Truth;log:forAll :c1, :c2, :c3, :p.
```

The query:

```
# File ontology1.query.n3
# rdfs ontology.
@prefix ont: <http://www.w3.org/2002/07/daml+oil#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
@prefix : <ontology#>.

# ?who rdfs:subClassOf :vertebrae.
# :spitsmuis rdfs:subClassOf ?who.
?w1 rdfs:subClassOf ?w2.
```

Paths in a graph

The axiom file:

```
# File path.axiom.n3
# definition of rdfs ontology
# modelled after: A Logical Interpretation of RDF
# Wolfram Conen, Reinhold Klapsing, XONAR IT and Information Systems
# and Software Techniques Velbert, Germany U Essen, D-45117

@prefix rdfs: <http://test#>.
@prefix : <ontology#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#>.

:m :n :o.
:a :b :c.
:c :d :e.
:e :f :g.
:g :h :i.
:i :j :k.
:k :l :m.
:o :p :q.
:q :r :s.
:s :t :u.
:u :v :w.
:w :x :z.
:z :f :a.

{?c1 ?p ?c2.} log:implies {?c1 :path ?c2.}.

{?c2 :path ?c3.?c1 :path ?c2.}  log:implies {?c1 :path ?c3.}.
```

The query:

```
# File path.query.n3
# rdfs ontology.

@prefix rdfs: <http://test#>.
@prefix : <ontology#>.
@prefix log: <http://www.w3.org/2000/10/swap/log#>.

#:a :path :a.
#?w :path :z.
```

```
?x :path ?y.
```

Logic gates

The axiom file:

```
definition of a half adder (from Wos Automated reasoning)

definition of a nand gate

gnand1(G,A),gnand2(G,B),gnand3(G,C),nand1(A,"false"),nand2(B,"false") :>
nand3(C,"true").
gnand1(G,A),gnand2(G,B),gnand3(G,C),nand1(A,"false"),nand2(B,"true") :>
nand3(C,"true").
gnand1(G,A),gnand2(G,B),gnand3(G,C),nand1(A,"true"),nand2(B,"false") :>
nand3(C,"true").
gnand1(G,A),gnand2(G,B),gnand3(G,C),nand1(A,"true"),nand2(B,"true") :>
nand3(C,"false").

definition of the connection between gates

nandnand2(G1,G2),gnand3(G1,X),gnand2(G2,X),nand3(X,V) :> nand2(X,V).
nandnand1(G1,G2),gnand3(G1,X),gnand1(G2,X),nand3(X,V) :> nand1(X,V).

The facts of the example:

gnand1(g1,a).
gnand2(g1,b).
gnand3(g1,a11).
nand1(a,"true").
nand2(b,"true").
nandnand1(g1,g3).
nandnand2(g1,g2).
nandnand2(g1,g9).

gnand1(g2,a).
gnand2(g2,a11).
gnand3(g2,a12).
nandnand1(g2,g4).

gnand1(g3,a11).
gnand2(g3,b).
gnand3(g3,a13).
nandnand2(g3,g4).
```

```
gnand1(g4,a12).
gnand2(g4,a13).
gnand3(g4,a14).
nandnand1(g4,g6).
nandnand1(g4,g5).

gnand1(g5,a14).
gnand2(g5,cin).
gnand3(g5,a15).
nand2(cin,"true").
nandnand1(g5,g8).
nandnand1(g5,g9).
nandnand2(g5,g6).

gnand1(g6,a14).
gnand2(g6,a15).
gnand3(g6,a16).
nandnand1(g6,g7).

gnand1(g7,a16).
gnand2(g7,a17).
gnand3(g7,sum).

gnand1(g8,a15).
gnand2(g8,cin).
gnand3(g8,a17).
nandnand2(g8,g7).

gnand1(g9,a15).
gnand2(g9,a11).
gnand3(g9,cout).
```

The queries:

```
nand3(sum,V).
nand3(cout,V).
```

Simulation of a flight reservation

I want to reserve a place in a flight going from Zaventem (Brussels) to Rio de Janeiro. The flight starts on next Tuesday (for simplicity only Tuesday is indicated as date). The respective **RDF** graphs are indicated G1, G2, etc… G1 is the system graph.
Namespaces are given by:
http://thesis/inf , abbreviated *inf*, the engine system space
http://thesis:flight_rules, abbreviated *frul*,the flight rules.
http://thesis/trust, abbreviated *trust*, the trust namespace.
http://flight_ontology/font, abbreviated *font*, the flight ontology space
http://international_air/air, abbreviated *air*, the namespace of the International Air company.
http://air_info/airinfo, abbreviated *ainf*, the namespace of the air information service.
These are non-existent namespaces created for the purpose of this simulation..

The process goes as follows:
1. start the query. The query contains an instruction for loading the flight rules.
2. Load the flight rules (module flight_rules.pro) in G2.
3. Make the query for a flight Zaventem – Rio.
4. The flight rules will direct a query to *air info*. This is simulated by loading the module air_info.pro in G3.
5. Air info answers yes, the company is International Air. The module air_info is deloaded from G3.
6. The flight rules will load the trust rules in G3: module trust.pro.
7. The flight rules ask whether the company International Air can be trusted.
8. The trust module answers yes, so a query is directed to International Air for commanding a ticket. This is simulated by loading the module International_air.pro in G4
9. International Air answers with a confirmation of the reservation.
10. The reservation info is signalled to the user. The user is informed of the results (query + info + proof).

The query:

```
prefix(inf, http://thesis/inf/).
prefix(font, http://flight_ontology/).
prefix(frul, http://thesis:flight_rules/).

inf:load(frul:flight_rules.pro).
font:reservation(R1).
R1(font:start,”Zaventem”,”Tuesday”).
```

```
R1(font:back, ”Zaventem”,”Thursday”).
R1(font:destination,”Rio”).
R1(font:price,inf:lower, ”3000”).
inf:deload(frul:flight_rules.pro).
```

The flight rules

```
prefix(inf, http://thesis/inf).
prefix(font, http://flight_ontology/font).
prefix(trust, http://thesis/trust).
prefix(ainf, http://air_info/airinfo).

inf:query(ainf:flight_info.pro),
X(font:start,Place,Day),
X(font:destination,Place1),
X(font:price,inf:lower,P),
font:flight(Y,X),
inf:end_query(ainf:flight_info.pro),
trust:trust(Y) ,
inf:query(Y),
font:ticket(X,I),
inf:end_query(Y),
inf:print(I),
inf:end_query(Y)
:> font:reservation(X).
```

flight_info.pro

```
prefix(inf, http://thesis/inf/).
prefix(font, http://flight_ontology/font/).
prefix(iair, http://international_air/air/).

font.flight(“intAir”,F716).
F716(font:flight_number,”F716”).
F716(font:destination,”Rio”).
F716(font:price,inf:lower, ”2000”).
```

Trust.pro

```
prefix(trust, http://thesis/trust).
trust:trust(“intAir”
```

Appendix 4. The handling of variables

I will explain here what the **RDFE**ngine does with variables in each kind of data structure: facts, rules and query.

There are four kinds of variables:
1) local universal variables
2) local existential variables
3) global universal variables
4) global existential variables

Blank nodes are treated like local existential variables.

The kernel of the engine, module **RDFE**ngine.hs, treats all variables as global existential variables. The scope of local variables is transformed to global by giving the variables a unique name. As the engine is an open world, constructive engine, universal variables do not exist (all elements of a set are never known). Thus a transformation has to be done from four kinds of variables to one kind.

Overview per structure:
1) facts:
   all variables are instantiated with a unique identifier.

2) rules:
   local means: within the rule.
   as in 1). If the consequent contains more variables than the antecedent then these must be instantiated with a unique identifier.

3) queries:
   local means: within a fact of the query; global means: within the query graph.
   In a query existential variables are not instantiated. Universal variables are transformed to existential variables.

Appendix 5. A theory of graph resolution

5.1. Introduction

In this section I present the complete series of definitions and lemmas.

5.2. Definitions

A *triple* consists of two labeled nodes and a labeled arc.

A *tripleset* is a set of triples.

A *graph* is represented by a set of triplesets.

A *rule* consists of antecedents and a consequent. The antecedents are a tripleset; the consequent is a triple.

Applying a rule *r* to a graph *G* in the closure process is the process of unifying the antecedents with all possible triples of the graph *G* while propagating the substitutions. For each successful unification the consequents of the rule is added to the graph with its variables substituted.

Applying a rule *r* to a subgraph *g* in the resolution process is the process of unifying the consequents with all possible triples of *g* while propagating the substitutions. For each successful unification the antecedents of the rule are added to the goallist of the resolution process (see the explanation of the resolution process).

A rule *r* is *valid* with respect to the graph *G* if its closure *G'* with respect to the graph *G* is a valid graph.

A graph *G* is *valid* if all the elements of its representation are triples.

The graph *G entails* the graph *T* using the rule *R* if *T* is a subgraph of the closure *G'* of the graph *G* with respect to the rule *R*.

The *closure G'* of a graph *G* with respect to a ruleset *R* is the result of the application of the rules of the ruleset *R* to the graph *G* giving intermediate graphs *Gi* till no more new triples can be generated. A graph may not contain duplicate triples.

The *closure G'* of a graph *G* with respect to a closure path is the resulting graph *G'* consisting of *G* with all triples added generated by the closure path.

A *solution to a query* with respect to a graph *G* and a ruleset *R* is a subgraph of the closure *G'* that unifies with the query.
A *solution set* is the set of of all possible solutions.

A *closure path* is a sequence of triples (g_i, r_i, c_i) where g_i is a subgraph of a graph G_i derived from the given graph *G*, r_i is a rule and c_i are the consequents of the rule with the substitution s_i.s_i is the substitution obtained trough matching a subgraph of G_i with the antecedents of the rule r_i.The tripleset c_i is added to G_i to produce the graph G_j. All triples in a closure path are grounded.

A *minimal closure path* with respect to a query is a closure path with the property that the graph produced by the path matches with the query and that it is not possible to diminish the length of the path i.e. to take a triple away.

The graph *generated* by a closure path consists of the graph *G* with all triples added generated by the rules used for generating the closure path.

A closure path *generates* a solution when the query matches with a subgraph of the graph generated by the closure path.

The *reverse path* of a closure path $(g_1, r_1, c_1),(g_2, r_2, c_2), \ldots,(g_n, r_n, c_n)$ is: $(c_n, r_n, g_n), \ldots (c_2, r_2, g_2),(c_1, r_1, g_1)$.

A *resolution path* is a sequence of triples (c_i, r_i, g_i) where c_i is subgraph that matches with the consequents of the rule r_i to generate g_i and a substitution s_i . However in the resolution path the accumulated substitutions $[s_i]$ are applied to all triplesets in the path.

A *valid* resolution path is a path where the triples of the query become grounded in the path after substitution or match with the graph *G*; or the resolution process can be further applied to the path so that finally a path is obtained that contains a solution. All resolution paths that are not valid are *invalid* resolution paths. The resolution path is *incomplete* if after substitution the triple sets of the path still contain variables. The resolution path is *complete* when, after substitution, all triple sets in the path are grounded.

A *final resolution path* is the path that rests after the goallist in the (depth first search) resolution process has been emptied. The path then consists of all the rule applications applied during the resolution process.

A *looping resolution path* is a path with a recurring triple in the sequence $(c_n, r_n, g_n), \ldots(c_x, r_x, g_x) \ldots(c_y, r_y, g_y)\ldots (c_2, r_2, g_2),(c_1, r_1, g_1)$ i.e. (c_x, r_x, g_x) is equal to some (c_y, r_y, g_y). The comparison is done after stripping of the level numbering.

The *resolution process* is the building of resolution paths. A *depth first search* resolution process based on backtracking is explained elsewhere in the text.

5.3. Lemmas

Resolution Lemma. There are three possible resolution paths:

1) a looping path
2) a path that stops and generates a solution
3) a path that stops and is a failure

Proof. Obvious.

Closure lemma. The reverse of a closure path is a valid, complete and final resolution path for a certain set of queries.

Proof. Take a closure path: $(g_1, r_1, c_1),(g_2, r_2, c_2), \ldots,(g_n, r_n, c_n)$. This closure path adds the triplesets $c_1, \ldots,c_n$ to the graph. All queries that match with these triples and/or triples from the graph *G* will be proved by the reverse path: $(c_n, r_n, g_n), \ldots (c_2, r_2, g_2), (c_1, r_1, g_1)$. The reverse path is complete because all closure paths are complete. It is final because the triple set *g1* is a subgraph of the graph *G* (it is the starting point of a closure path). Take now a query as defined above: the triples of the query that are in *G* do of course match directly with the graph. The other ones are consequents in the closure path and thus also in the resolution path. So they will finally match or with other consequents or with triples from *G*.

Final Path Lemma: The reverse of a final path is a closure path. All final paths are complete and valid.

Proof: A final resolution path is the resolution path when the goal list has been emptied and a solution has been found.
Take the resolution path:
$(c_1, r_1, g_1), (c_2, r_2, g_2), \ldots (c_{n-1}, r_{n-1}, g_{n-1}), (c_n, r_n, g_n)$
This path corresponds to a sequence of goals in the resolution process:
$c_1, c_{2,\ldots,} c_{n-1}, c_n$. One moment or another during the process these goals are in the goallist. Goals generate new goals or are taken from the goallist when they are grounded. This means that if the goallist is empty all variables must have been substituted by URI's or literals. But perhaps a temporary variable could exist in the sequence i.e. a variable that appears in the antecedents of a rule *x*, also in the

consequents of a rule y but that has disappeared in the antecedents of rule *y*. However this means that there is a variable in the consequents of rule y that is not present in the antecedents what is not permitted.
Be $(c_n, r_n, g_n), \ldots (c_2, r_2, g_2), (c_1, r_1, g_1)$ a final resolution path. I already proved in the first part that it is grounded because all closure paths are grounded. g_1 is a subgraph of *G* (if not, the triple (c_1, r_1, g_1) should be followed by another one). c_n matches necessarily with the query. The other triples of the query match with the graph *G* or match with triples of c_i, c_j etc.. (consequents of a rule). The reverse is a closure path. However this closure path generates all triples of the query that did not match directly with the graph *G*. Thus it generates a solution.

Looping Lemma I. If a looping resolution path contains a solution, there exists another non looping resolution path that contains the same solution.

Proof. All solutions correspond to closure paths in the closure graph *G'* (by definition). The reverse of such a closure path is always a valid resolution path and is non-looping.

Looping Lemma II. Whenever the resolution process is looping, this is caused by a looping path if the closure of the graph *G* is not infinite.
Proof. Looping means in the first place that there is never a failure i.e. a triple or tripleset that fails to match because then the path is a failure path and backtracking occurs. It means also triples continue to match in an infinite series. There are two sublemmas here:
Sublemma 1. In a looping process at least one goal should return in the goallist.
*Proof.*Can the number of goals be infinite? There is a finite number of triples in *G*, a finite number of triples that can be generated and a finite number of variables. This is the case because the closure is not infinite. So the answer is no. Thus if the process loops (ad infinitum) there is a goal that must return at a certain moment.
Sublemma 2. In a looping process a sequence (rule *x*) goal *x* must return. Indeed the numbers of rules is not infinite; neither is the number of goals; if the same goal returns infinitly, sooner or later it must return as a deduction of the same rule *x*.

Infnite Looping Path Lemma. If the closure graph *G'* is not infinite and the resolution process generates an infinite number of looping paths then this generation will be stopped by the anti-looping technique and the resolution process will terminate in a finite time.

Proof. The finiteness of the closure graph implies that the number of labels and arcs is finite too.
Suppose that a certain goal generates an infinite number of looping paths.

This implies that *the breadth* of the resolution tree has to grow infinitely. It cannot grow infinitely in one step (nor in a finite number of steps) because each goal only generates an finite number of children (the database is not infinite). This implies that also *the depth* of the resolution tree has to grow infinitely. This implies that the level of the resolution process grows infinitely and also then the number of entries in the oldgoals list.
Each growth of the depth produces new entries in the oldgoals list. (In this list each goal-rule combination is new because when it is not, a backtrack is done by the anti-looping mechanism). In fact with each increase in the resolution level an entry is added to the oldgoals list.
Then at some point the list of oldgoals will contain all possible rule-goals combinations and the further expansion is stopped.
Formulated otherwise, for an infinite number of looping paths to be materialized in the resolution process the oldgoals list should grow infinitely but this is not possible.
Addendum: be *maxG* the maximum number of combinations (goal, rule). Then no branch of the resolution tree will attain a level deeper than *maxG*. Indeed at each increase in the level a new combination is added to the oldgoals list.

Infinite Lemma. For a closure to generate an infinite closure graph *G'* it is necessary that at least one rule adds new labels to the vocabulary of the graph.

Proof. Example: *{:a log:math :n } → {:a log:math :n+1)*. The consequent generated by this rule is each time different. So the closure graph will be infinite. Suppose the initial graph is: *:a log:math :0*. So the rule will generate the sequence of natural numbers.
If no rule generates new labels then the number of labels and thus of arcs and nodes in the graph is finite and so are the possible ways of combining them.

Duplication Lemma. A solution to a query can be generated by two different valid resolution paths. It is not necessary that there is a cycle in the graph *G'*.

Proof. Be $(c_1, r_1, g_1), (c_2, r_2, g_2), (c_3, r_3, g_3)$ and $(c_1, r_1, g_1), (c_3, r_3, g_3), (c_2, r_2, g_2)$ two resolution paths.Let the query *q* be equal to *c1*. Let *g1* consist of *c2* and *c3* and let *g2* and *g3* be subgraphs of *G*. Then both paths are final paths and thus valid (Final Path Lemma).

Failure Lemma. If the current goal of a resolution path (the last triple set of the path) can not be unified with a rule or facts in the database, it can have a solution, but this solution can also be found in a final path.

Proof. This is the same as for the looping lemma with the addition that the reverse of a failure path cannot be a closure path as the last triple set is not in the graph *G* (otherwise there would be no failure).

Substitution Lemma I. When a variable matches with a **URI** the first item in the substitution tuple needs to be the variable.

Proof. 1) The variable originates from the query.

a) The triple in question matches with a triple from the graph *G*. Obvious, because a subgraph matching is being done.
b) The variable matches with a ground atom from a rule.
Suppose this happens in the resolution path when using rule r_1:
$(c_n, r_n, g_n), \ldots (c_2, r_2, g_2), (c_1, r_1, g_1)$
The purpose is to find a closure path:
$(g_1, r_1, c_1), \ldots, (g_x, r_x, c_x), \ldots, (g_n, r_n, c_n)$ where all atoms are grounded.
When rule r_1 is used in the closure the ground atom is part of the closure graph, thus in the resolution process , in order to find a subgraph of a closure graph, the variable has to be substituted by the ground atom.

2) A ground atom from the query matches with a variable from a rule. Again, the search is for a closure path that leads to a closure graph that contains the query as a subgraph. Suppose a series of substitutions:
(v_1, a_1) (v_2, a_2) where a_1 originates from the query corresponding to a resolution path: $(c_1, r_1, g_1), (c_2, r_2, g_2), (c_3, r_3, g_3)$. Wherever v_1 occurs it will be replaced by a_1. If it matches with another variable this variable will be replaced by a_1 too. Finally it will match with a ground term from *G'*. This corresponds to a closure path where in a sequence of substitutions from rules each variable becomes replaced by a_1, an atom of *G'*.

3) A ground atom from a rule matches with a variable from a rule or the inverse. These cases are exactly the same as the cases where the ground atom or the variable originates from the query.

Substitution Lemma II. When two variables match, the variable from the query or goal needs to be ordered first in the tuple.

Proof. In a chain of variable substitution the first variable substitution will start with a variable from the query or a goal. The case for a goal is similar to the case for a query. Be a series of substitutions:
(v_1, v_2) $(v_2, v_x) \ldots (v_x, a)$ where *a* is a grounded atom.
By applying this substitution to the resolution path all the variables will become equal to *a*. This corresponds to what happens in the closure path where the ground atom a will replace all the variables in the rules.

Solution Lemma I: A solution corresponds to an empty closure path (when the query matches directly with the database) or to one or more minimal closure paths.

Proof: there are two possibilities: the triples of a solution either belong to the graph *G* or are generated by the closure process. If all triples belong to *G* then the closure path is empty. If not there is a closure path that generates the triples that do not belong to *G*.

Solution Lemma II: be *C* the set of all closure paths that generate solutions. Then there exist matching resolution paths that generate the same solutions.

Proof: Be $(g_1, r_1, c_1),\ldots, (g_x, r_x, c_x),\ldots, (g_n, r_n, c_n)$ a closure path that generates a solution.
Let $((c_n, r_n, g_n), \ldots (c_2, r_2, g_2), (c_1, r_1, g_1)$ be the corresponding resolution path. r_n applied to c_n gives g_n. Some of the triples of g_n can match with *G*, others will be deduced by further inferencing (for instance one of them might be in c_{n-1}). When c_n was generated in the closure it was deduced from g_n. The triples of g_n can be part of *G* or they were derived with rules $r_1,\ldots, r_{n-2}$. If they were derived then the resolution process will use those rules to inference them in the other direction. If a triple is part of *G* in the resolution process it will be directly matched with a corresponding triple from the graph *G*.

Solution Lemma III. The set of all valid solutions to a query *q* with respect to a graph *G* and a ruleset *R* is generated by the set of all valid resolution paths, however with elimination of duplicates.

Proof. A valid resolution path generates a solution. Because of the completeness lemma the set of all valid resolution paths must generate all solutions. But the duplication lemma states that there might be duplicates.

Solution Lemma IV. A complete resolution path generates a solution and a solution is only generated by a complete, valid and final resolution path.
Proof. → : be $(c_n, r_n, g_n),\ldots, (c_x, r_x, g_x),\ldots, (c_1, r_1, g_1)$ a complete resolution path. If there are still triples from the query that did not become grounded by 1) matching with a triple from *G* or 2) following a chain $(c_n, r_n, g_n),\ldots, (c_1, r_1, g_1)$ where all gn match with *G*, the path cannot be complete. Thus all triples of the query are matched and grounded after substitution, so a solution has been found.
← : be $(c_n, r_n, g_n),\ldots, (c_x, r_x, g_x),\ldots, (c_1, r_1, g_1)$ a resolution path that generates a solution. This means all triples of the query have 1) matched with a triple from *G* or 2) followed a chain $(c_n, r_n, g_n),\ldots, (c_1, r_1, g_1)$ where g_n matches with a subgraph from *G*. But then the path is a final path and is thus complete (Final Path Lemma).

Completeness Lemma. The depth first search resolution process with exclusion of invalid resolution paths generates all possible solutions to a query *q* with respect to a graph *G* and a ruleset *R*.

Proof. Each solution found in the resolution process corresponds to a closure path. Be *S* the set of all those paths. Be *S1* the set of the reverse paths which are resolution paths (Valid Path Lemma). The set of paths generated by the depth first search resolution process will generate all possible combinations of the goal with the facts and the rules of the database thereby generating all possible final paths.
It is understood that looping paths are stopped.

Monotonicity Lemma. Given a merge of a graph *G1* with rule set *R1* with a graph *G2* with rule set *R2*; a query *Q* with solution set *S* obtained with graph *G1* and rule set *R1* and a solution set *S'* obtained with graph *G2* and rule set *R2*. Then *S* is contained in *S'*.

Proof. Be *G1'* the closure of *G1*. The merged graph *Gm* has a rule set *Rm* that is the merge of rule sets *R1* and *R2*. All possible closure paths that produced the closure *G1'* will still exist after the merge. The closure graph *G1'* will be a subgraph of the graph *Gm*. Then all solutions of solution set *S* are still subgraphs of *Gm* and thus solutions when querying against the merged graph *Gm*.

5.4.An extension of the theory to variable arities

5.4.1. Introduction

A **RDF** triple can be seen as a predicate with arity two. The theory above was exposed for a graph that is composed of nodes and arcs. Such a graph can be represented by triples. These are not necessarily **RDF** triples. The triples consist of two node labels and an arc label. This can be represented in predicate form where the arc label is the predicate and the nodes are its arguments.
This theory can easily be extended to predicates with other arities.
Such an extension does not give any more semantic value. It does permit however to define a much more expressive syntax.

5.4.2. Definitions

A *multiple of arity n* is a predicate *p* also called arc and *(n-1)* nodes also called points. The points are connected by an (multi-headed) arc. The points and arcs are labeled and no two labels are the same.
A *graph* is a set of multiples.

Arcs and nodes are either variables or labels. (In internet terms labels could be **URI**'s or literals).
Two multiples *unify* when they have the same arity and their arcs and nodes unify.
A variable unifies with a variable or a label.
Two labels unify if they are the same.
A *rule* consists of *antecedents* and a *consequent.* Antecedents are multiples and a consequent is a multiple.
The *application of a rule* to a graph *G* consists in unifying the antecedents with multiples from the graph and, if successful, adding the consequent to the graph.

5.4.3. Elaboration

The lemmas of the theory explained above can be used if it is possible to reduce all multiples to triples and then proof that the solutions stay the same.
Reduction of multiples to triples:
Unary multiple: In a graph *G a* becomes *(a, a, a)* .In a query *a* becomes *(a, a, a).*
These two clearly match.
Binary multiple: *(p, a)* becomes *(a, p, a)*. The query *(?p, ?a)* corresponds with *(?a, ?p, ?a).*
Ternary multiple: this is a triple.
Quaternary multiple: *(p, a, b, c)* becomes:
{(x, p, a), (x, p, b), (x,p,c)} where *x* is a unique temporary label.
The query *(?p, ?a, ?b, ?c)* becomes:
{(y, ?p, ?a),(y, ?p, ?b),(y, ?p, ?c)}.
It is quit obvious that this transformation does not change the solutions.
Other arities are reduced in the same way as the quaternary multiple.

5.5.An extension of the theory to typed nodes and arcs

5.5.1. Introduction

As argued also elsewhere (chapter 6) attributing types to the different syntactic entities i.e. nodes and arcs could greatly facilitate the resolution process.
Matchings of variables with nodes or arcs that lead to failure because the type is not the same can thus be impeded.

5.5.2. Definitions

A *sentence* is a sequence of *syntactic entities.* A syntactic entity has a *type* and a *label.* Example: verb/walk.
Examples of syntactic entities: verb, noun, article, etc… A syntactic entity can also be a variable. It is then preceded by an interrogation mark.

A *language* is a set of sentences.
Two sentences *unify* if their entities unify.
A *rule* consists of *antecedents* that are sentences and a *consequent* that is a sentence.

5.5.3. Elaboration

An example: *{(noun/?noun, verb/walks)} → {(article/?article,noun/ ?noun, verb/walks)}*
In fact what is done here is to create multiples where each label is tagged with a type. Unification can only happen when the types of a node or arc are equal besides the label being equal.
Of course sentences are multiples and can be transformed to triples.
Thus all the lemmas from above apply to these structures if solutions can be defined to be subgraphs of a closure graph. .
This can be used to make transformations e.g.
(?ty/label,object/?ty1) → (par/'(', object/?ty1).
Parsing can be done:
(?/charIn ?/= ?/'(') → (?/charOut ?/= par/'(')

Appendix 6. Abbreviations of namespaces in Notation 3

In N3 **URI**'s can be indicated in a variety of different ways. A **URI** can be indicated by its complete form or by an abbreviation. These abbreviations have practical importance and are intensively used in the testcases.

In what follows the first item gives the complete form; the others are abbreviations.

- *<http://www.w3.org/2000/10/swap/log#log:forAll>* : this is the complete form of a **URI** in Notation 3. This resembles very much the indication of a tag in an **HTML** page.
- ***xxx:*** : a label followed by a ':' is a prefix. A prefix is defined in N3 by the @prefix instruction :
  @prefix ont: <http://www.daml.org/2001/03/daml-ont#>.
  This defines the prefix ont: . After the label follows the **URI** represented by the prefix.
  So *ont:TransitiveProperty* is in full form
  <http://www.daml.org/2001/03/daml-ont#TransitiveProperty> .
- *<#param>* : here only the tag in the **HTML** page is indicated. To get the complete form the default **URL** must be added.the complete form is :
  <URL_of_current_document#param>.
  The default **URL** is the **URL** of the current document i.e. the document that contains the description in Notation 3.
- <> or <#>: this indicates the **URI** of the current document.
- **:** : a single colon is by convention referring to the current document. However this is not necessarily so because this meaning has to be declared with a prefix statement :
  @prefix ***: <#>.***