Summary

The semantic web is a vision about the future of the World Wide Web brought forward by the inventor of the web, Tim Berners-Lee. It is not an utopic vision but more the feeling that the web has created enormous possibilities and that the right thing to do is to make use of these possibilities. In this thesis an insight will be given into the "why" and "how" of the semantic web. The mechanisms that exist or that are being developed are explained in detail: **XML, RDF,** rdfschema, SweLL, proof engines and trust mechanisms. The layered model that structures and organizes these mechanisms is explained:see fig.1.

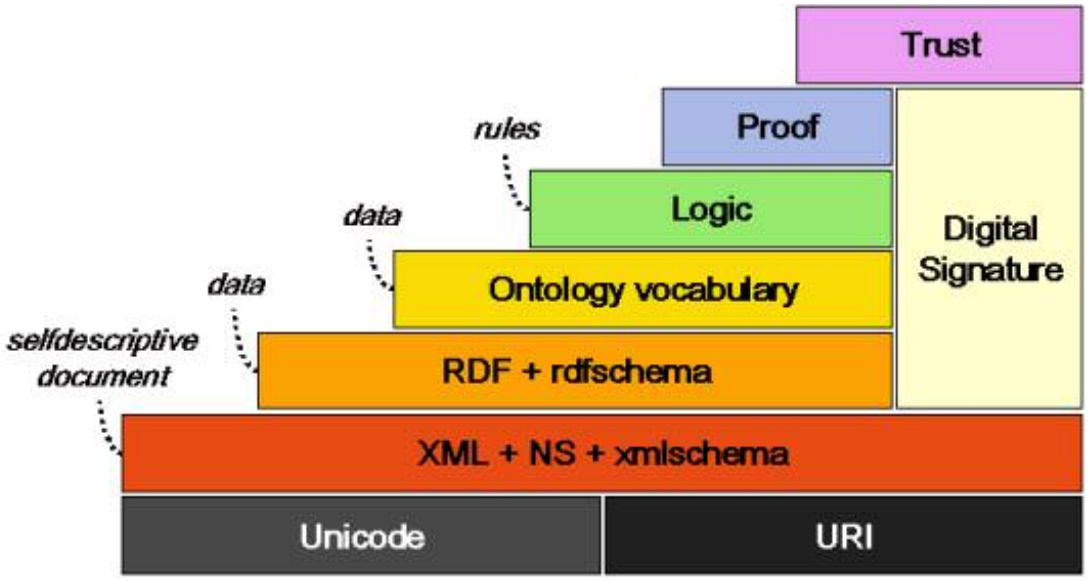


Fig. 1. The layered model of the semantic web after Tim Berners-Lee.

A parser and a proof engine based on Notation 3 , an alternative syntax for **RDF**, were developed and their mechanisms are described in detail. The basic resolution theory upon which the engine is based is explained in detail. Adapatability and evolvability were two of the main concerns in developing the engine. Therefore the engine is fed by *metadata* composed of rules and facts in Notation 3: see fig.2.

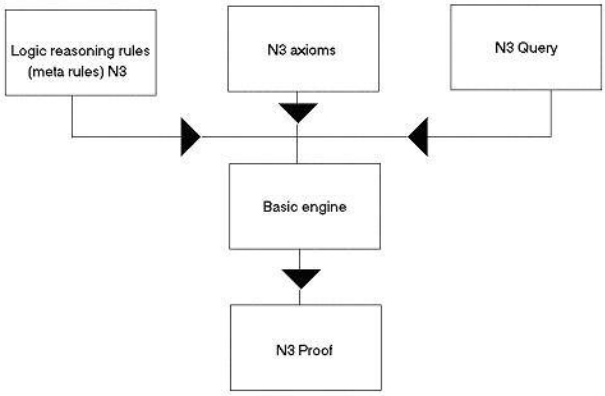


Fig.2 The structure of the inference engine. Input and output are in Notation 3.

The kernel of the engine, the basic engine, is kept as small as possible. Ontological or logic rules and facts are laid down in the set of metarules that govern the behaviour of the engine. In order to implement the owl ontology, freshly developed by the Ontology Workgroup of the W3C an experiment with typing has been done. By using a *typed system* restrictions can be applied to the ontological concepts. The typing also reduces the combinatorial explosion.

An executable specification of the engine was made in Haskell 98 (Hugs platform on Windows).

Besides this metadata the engine is fed with an axiom file (the facts and rules comparable to a Prolog program) and a query file (comparable to a Prolog query). The output is in the same format as the input so that it can serve again as input for the engine.

As the engine is based on logic and resolution, a literature study is included that gives an overview of theorem provers (or automated reasoning) and of the most relevant kinds of logic. This study was the basis for the insight in typing mechanisms.

The conclusion

The standardisation of the Semantic Web and the development of a standard ontology and proof engines that can be used to establish trust on the web is a huge challenge but the potential rewards are huge too. The computers of companies and citizens will be able to make complex completely automated interactions freeing everybody from administrative

and logistic burdens. A lot of software development remains to be done and will be done by enthousiastic software engineers.

Existing inference engines

CWM – Euler – other??? Sw.phpapp.org

The semantic web

By the advent of the internet a mass communication medium has become available. One of the most important and indeed revolutionary characteristics of the internet is the fact that now everybody is connected with everybody, citizens with citizens, companies with companies and citizens with companies, government etc... In fact now **the** global village exists. This interconnection creates astounding possibilities of which only few are used today. The internet serves mainly as a vehicle for hypertext. These texts with high semantic value for humans have little semantic value for computers.
The problem always with interconnection of companies is the specificity of the tasks to perform. EDI was an effort to define a framework that companies could use to communicate with each other. Other efforts have been done by standardizing XML-languages (eg. ebXML). At the current moment an effort endorsed by large companies is underway: web services.
The interconnection of all companies and citizens one with another creates the possibility of automating a lot of transactions that are now done manually or via specific and dedicated automated systems. It should be clear that separating the common denominator from all the efforts mentioned higher, standardizing it so that it can be reused a million times certainly has to be interesting. Of course standardisation may not develop into bureaucracy impeding further developments. But e.g. creating 20 times the same program with different programming languages does not seem very interesting either, except if you can leave the work to computers and even then, a good use should be made of computers.
If every time two companies connect to each other for some application they have to develop a framework for that application then the efforts to develop all possible applications become humongous. Instead a general system can be developed based on inference engines and ontologies. The mechanism is as follows: the interaction between the communicating partners to achieve a certain goal is laid down into facts and rules using a common language to describe those facts and rules where the flexibility is provided by the fact that the common language is in fact a series of languages and tools including in the semantic web vision: XML, RDF,

RDFS, DAML+OIL, SweLL, owl(see further). To achieve automatic interchange of information ontologies play a crucial role; as a computer is not able to make intelligent guesses to the meaning of something as humans do, the meaning of something (i.e. the semantics) have to be defined in terms of computer actions. A computer agent recieves a communication from another agent. It must then be able to transform that communication into something it understands i.e. that it can interpret and act upon. The word “transform” means that eventually the message may arrive in a different ontology than the one used by the local client but necessarily a transformation to his own ontology must be possible. Eventually an agreement between the two parties for some additional, non-standard ontologies has to be made for a certain application.
It is supposed that the inference engine has enough power to deal with all (practically all) possible situations.
Then their might be the following scheme for an application using the tecnology discussed and partially implemented within this thesis:
Lay down the rules of the application in Notation 3. One partner then sends a query to another partner. The inference engine interprets this query thereby using its set (sets) of ontological rules and then it produces an answer. The answer indeed might consist of statements that will be used by another soft to produce actions within the recieving computer. What has to be done then? Establishing the rules and making an interface that can transform the response of the engine into concrete actions.
The semantics of this all [USHOLD] lies in the interpretation by the inference engine of the ontological rule sets that it disposes of and their specific implementation by the engine and in the actions performed by the interface as a consequence of the engine’s responses. Clearly the actions performed after a conclusion from the engine give place to a lot of possible standardisation. (A possible action might be: sending a SOAP message. Another might be: sending a mail).
What will push the semantic web are the enormous possibilities of automated interaction created by the sole existence of the internet between communication partners: companies, government, citizens. To say it simply: the whole thing is too interesting not to be done!!!
The question will inevitable be raised whether this development is for the good or the bad. The hope is that a further, perhaps gigantesque, development of the internet will keep and enhance its potentialities for defending and augmenting human freedom.

A case study

Fig.1 gives a schematic view of 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 France 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.

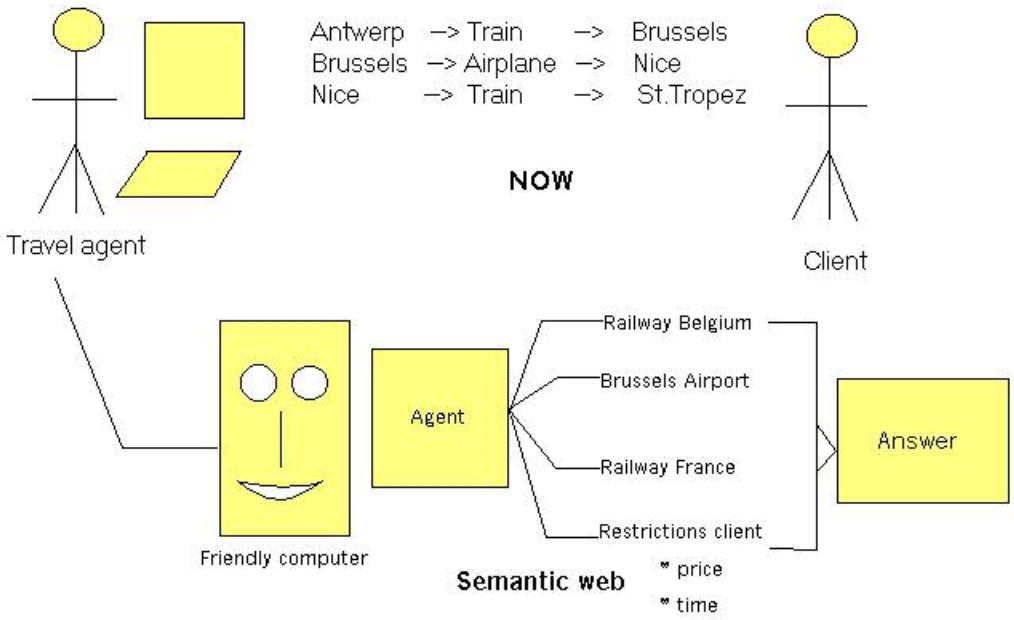


Fig.1. A semantic web case study.

He 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 to some alternatives. The client will come back to make a final decision once the travel agent has adverted 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 om 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 should exist. In the computer of the travel agent resides a semantic web agent who disposes of the complete range of necessary layers: XML, RDF, RDFS, ontological layer, logic layer, proof layer and trust layer (this will be explained in more detail later). 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 standardized query format for the semantic web agent. The agent consult his rule database (in Notation3: see further). 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: http://www.agfa.com/w3c/euler/graph.axiom.n3). 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 recieved 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 saved an important amount of his time.

Conclusions:

That a realisation of such a 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 reasonings of this case study (path determination, itinerary scheduling). All this rules have then 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 softwares that permit the client to manage his travel needs. But all those software have to be fed with information obtained by a variety of means, practically always manually.

The WorldWide Web Consortium – W3C

[W3SCHOOLS]

The World Wide Web (WWW) began as a project of Tim Berners-Lee at the European Organization for Nuclear Research (CERN) [TBL]. W3C was created in 1994 as a collaboration between the Massachusetts Institute of Technology (MIT) and the European Organization for Nuclear Research (CERN), with support from the U.S. Defense Advanced Research Project Agency (DARPA) and the European Commission.The director of the WorldWide Web is Tim Berners-Lee. W3C also coordinates its work with many other standards organizations such as the Internet Engineering Task Force, the Wireless Application Protocols (WAP) Forum and the Unicode Consortium.
W3C is hosted by three universities: Massachusetts Institute of Technology in the U.S., The French National Research Institute in Europe and Keio University in Japan.
[http://www.w3.org/Consortium/]
W3C's long term goals for the Web are:

1. Universal Access: To make the Web accessible to all by promoting technologies that take into account the vast differences in culture, languages, education, ability, material resources, and physical limitations of users on all continents;
2. Semantic Web : To develop a software environment that permits each user to make the best use of the resources available on the Web;
3. Web of Trust : To guide the Web's development with careful consideration for the novel legal, commercial, and social issues raised by this technology.

Design Principles of the Web:

The Web is an application built on top of the Internet and, as such, has inherited its fundamental design principles.

1. *Interoperability*: Specifications for the Web's languages and protocols must be compatible with one another and allow (any) hardware and software used to access the Web to work together.
2. *Evolution*: The Web must be able to accommodate future technologies. Design principles such as simplicity, modularity, and extensibility will increase the chances that the Web will work with emerging technologies such as mobile Web devices and digital television, as well as others to come.
3. *Decentralization*: Decentralization is without a doubt the newest principle and most difficult to apply. To allow the Web to "scale"

to worldwide proportions while resisting errors and breakdowns, the architecture(like the Internet) must limit or eliminate dependencies on central registries.

The work is divided into 5 domains:
Architecture Domain :
The Architecture Domain develops the underlying technologies of the Web.
Document Formats Domain :
The Document Formats Domain works on formats and languages that will present information to users with accuracy, beauty, and a higher level of control.
Interaction Domain:
The Interaction Domain seeks to improve user interaction with the Web, and to facilitate single Web authoring to benefit users and content providers alike.
Technology and Society Domain :
The W3C Technology and Society Domain seeks to develop Web infrastructure to address social, legal, and public policy concerns.
Web Accessibility Initiative (WAI):
W3C's commitment to lead the Web to its full potential includes promoting a high degree of usability for people with disabilities. The Web Accessibility Initiative (WAI), is pursuing accessibility of the Web through five primary areas of work: technology, guidelines, tools, education and outreach, and research and development.

The most important work done by the W3C is the development of "Recommendations" that describe communication protocols (like HTML and XML) and other building blocks of the Web.
Each W3C Recommendation is developed by a work group consisting of members and invited experts.
W3C Specification Approval Steps:

- W3C receives a Submission
- W3C publishes a Note
- W3C creates a Working Group
- W3C publishes a Working Draft
- W3C publishes a Candidate Recommendation
- W3C publishes a Proposed Recommendation
- W3C publishes a Recommendation

<u>Why does the semantic web need inference engines?</u>

Mister Reader is interested in a book he has seen from a catalogue on the internet from the company GoodBooks. He fills in the form for the

command mentioning that he is entitled to become a reduction. Now GoodBooks need to do two things first: see to it that mr. Reader is who he claims to be and secondly verify if he is really entitled to a reduction by checking the rule-database where reductions are defined. The secret key of mr. Reader is certified by CertificatesA. CertificatesA is certified by CertificatesB. CertificatesB is a trusted party. Now certification is known to be an owl:transitiveProperty (for owl see further) so the inference engine of GoodBooks concludes that mr Reader is really mr Reader. Indeed a transitive property is defined by: if from a follows b and from b follows c then from a follows c. Thus if X is certified by A and A is certified by B then X is certified by B. Now the reduction of mr Reader needs to be checked. Nothing is found in the database, so a query is sent to the computer of mr Reader asking for the reason of his reduction. As an answer the computer of mr Reader sends back: I have a reduction because I am an employee of the company BuysALot. This “proof” has to be verified. A rule is found in the database stating that employees of BuysALot have indeed reductions. But is mr Reader an emplyee? A query is send to BuysALot asking whether mr Reader is an employee. The computer of BuysALot does not know the notion employee but finds that employee is daml:equivalentTo worker and that mr Reader is a worker in their company so they send back an affirmative answer to GoodBooks. GoodBooks again checks the secret key of BuysALot and now can conclude that mr Reader is entitled to a reduction. The book will be sent. Now messages go away to the shipping company where other engines start to work, the invoice goes to the bank of mr Reader whose bank account is obtained from his computer while he did not fill in anything in the form etc... Finally mr Reader recieves his book and the only thing he did do was to check two boxes.

The layers of the semantic web

Fig.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

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

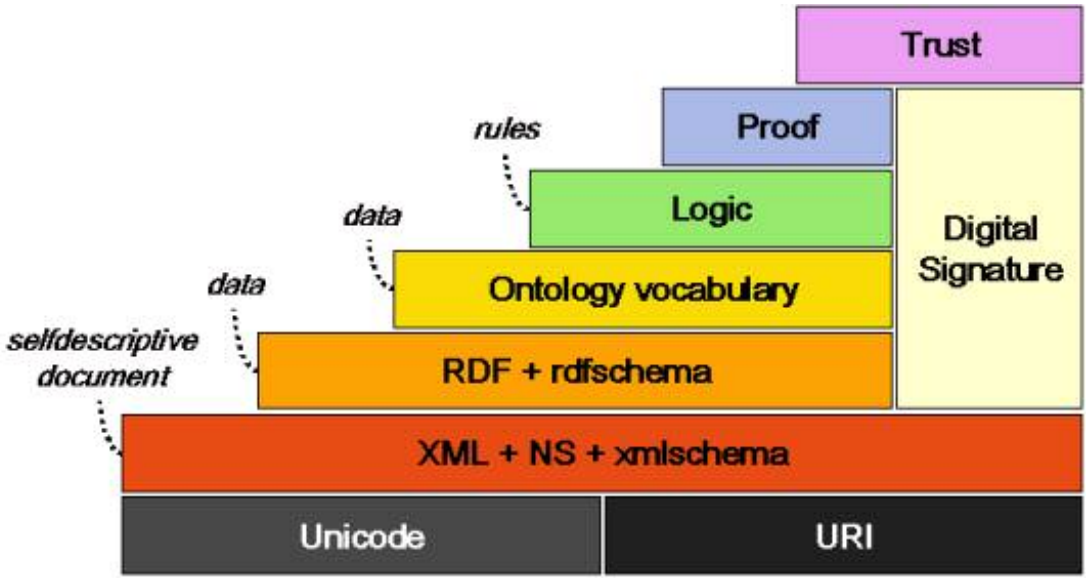


Fig.2 The layers of the semantic web [Berners-Lee].

Unicode codes codes the characters of all the major languages in use today.[*http://www.unicode.org/unicode/standard/principles.html*]. There are 3 formats for encoding unicode characters. These formats are convertible one into another.

1) in UTF-8 character size is variable. Ascii characters remain unchanged when transformed to UTF-8.
2) In UTF-16 the most heavily used characters use 2 bytes, while others use 4 bytes.
3) In UTF-32 all characters are encoded in 4 bytes.

URI's are Universal Resource Indicators. With a uri some"thing" is indicated in a unique and universal way. An example is an indication of an e-mail by the concatenation of email address and date and time.

Layer 2

XML stands for eXtensible Markup Language.
XML is a meta-language that permits to develop new languages following XML syntax and semantics. In order not to confuse the notions of different languages each language has a unique namespace tha is defined by a URI. This gives the possibility to mix different languages in one XML object.

Xmlschema gives the possibility of describing a developed language: its elements and the restrictions that must be applied to them.

XML is a basic tool for the exchange of information between communicating partners on the internet. The communication is by way of a selfdescriptive document.

Layer 3

The first two layers consist of basic internet technologies. With layer 3 starts the semantic web. RDF has as main goal the description of data. RDF stands for Resource Description Framework.
The basic principle is that information is expressed in triples: subject – property – object e.g. person – name – Naudts. That is the basic semantics of RDF. The syntax can be XML, Notation 3 or something else (see further).
Rdfsschema 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 definitions of rdfschema 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

In the case study the use of rulesets was mentioned. For expressing rules a logic layer is needed. An experimental logic layer exists [SWAP/CWM].
Layer 6

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 actif 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 thesite. 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 N3Engine developed as part of this thesis.

Layer 7

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 thet 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.
The *trust policy* is laid down in a "facts and rules" database (e.g. in Notation 3). This database is used by an inference engine like N3Engine. A user defines his policy using a GUI that produces an N3 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 that this whole layered builing has as purpose to implement trust on the internet. Indeed it is necessary for implementing trust but, once the pyramid of fig. 2 comes into existence, on top of it all kind of applications can be build.

Layer 8

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.

<u>A web of trust</u>

It might seem strange to speak first of the highest layer. The reason is that understanding the necessities of this layer can give the insight as to the "why?" of the other layers. To realise a web of trust all the technologies of the underlying layers are necessary.

Basic mechanisms

[SCHNEIER].
Historically the basic idea of cryptography was to *encrypt* a text using a *secret key*. The text can then only be decrypted by someone disposing of the secret key. The famous *Caesar cipher* was just based on displacing all the characters in the alphabet e.g. "a" becomes "m", "b" becomes "n" etc... Based also on a secret key is the *DES algorithm*. In this algorithm, based on the secret key, the text is transformed in an encrypted text by complex manipulationsof the text. As the reader might

guess this is a lot more complicated than the Caesar cipher and still a good cryptography mechanism. A revolution was the invention of *trap-door one-way functions* by Rivest, Shamir and Adleman in 1977. Their first algorithm was based on properties of prime numbers. [course on discrete mathematics]. A text is encrypted by means of a public key and only he who disposes of the private key (the trap-door) kan decipher the text.
Combined with hashing this gives the *signature* algorithms. Hashing means reducing the information content of a file to a new file of fixed length e.g; 2 Kilobytes. So a document of 6 Mega is reduced to 2 Kilobytes; one of 100 bytes is also reduced to 2 Kilobytes. The most important feature of hashing is that it is practically impossible given a document with its hashed version to produce a second document with the same hashing. So a hash constitutes a *fingerprint* of a document.
Fig. 1 show the mechanism of *digital signature.* The sender of a document generates a hash of his document. Then he encrypts this hash with his private key. The document together with the encrypted hash is send to the reciever. The reciever decrypts the hash with the public key of the sender. He then knows that the hash is produced by the owner of the public key. His confidence in the ownership of the public key is generated either by a PKI or by a web of trust (see further). The the reciever produces a hash of the original document. If his hash is the same as the hash that has been sent to him then he knows that the document has not been changed while travelling to him. Thus the integrity is safeguarded.

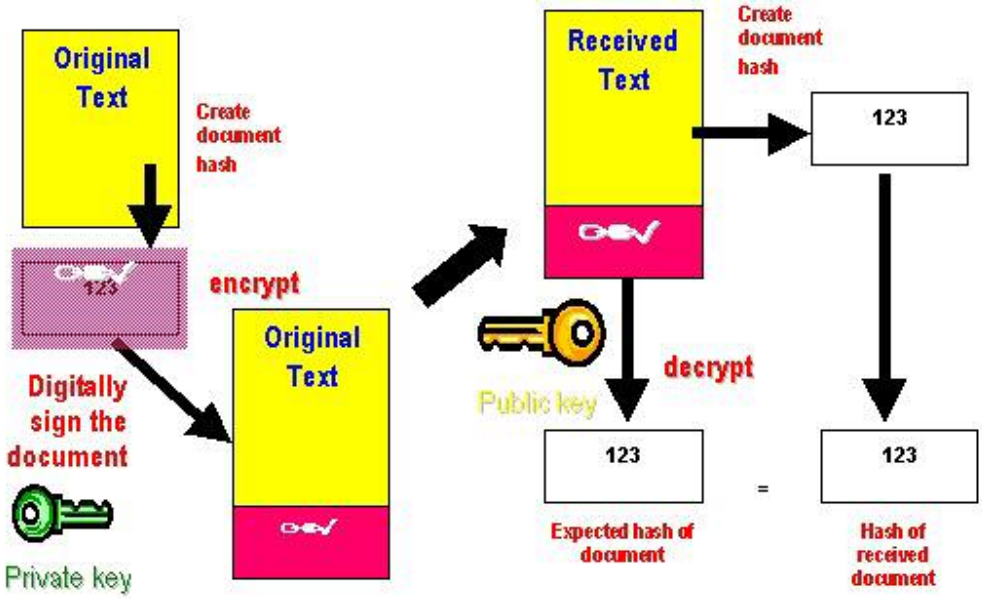


Fig. 1. The mechanism of digital signature.

In general following characteristics are important for security:

1) Privacy: your communication has only been seen by the persons that are authorised to see it.
2) Integrity: you are sure that your communication has not been tampered with.
3) Authenticity: the reciever is sure that the text he recieves has been send by you and not by an imposter.
4) Non repudiation: someone send you a text but afterwards denies that he has sent it. However the text was signed with his private key so the odds are against him.
5) Autorisation: the person who accesses a database is he really authorised to do so?

PKI or Public Key Infrastructure

As was said higher: how do you know that the public key you use does really belong to the person you assume he belongs to? One solution to this problem is a public key infrastructure or PKI. A user of ompany A who wants to obtain a private - public key pair applies for it at his local *RA (Registration Authority)*. The RA send a demand for a key pair to the *CA (Certification Authority)*. The user then recieves a *Certificate* from the CA. This certificate is signed with the *root (private) key* of the CA. The public key of the CA is a well known key that can be found on the internet. When I send a signed document to someone I send my certificate also. The reciever can then verify that my public key was issued to me by the CA by decrypting the signature of the certificate with the root public key of the CA.

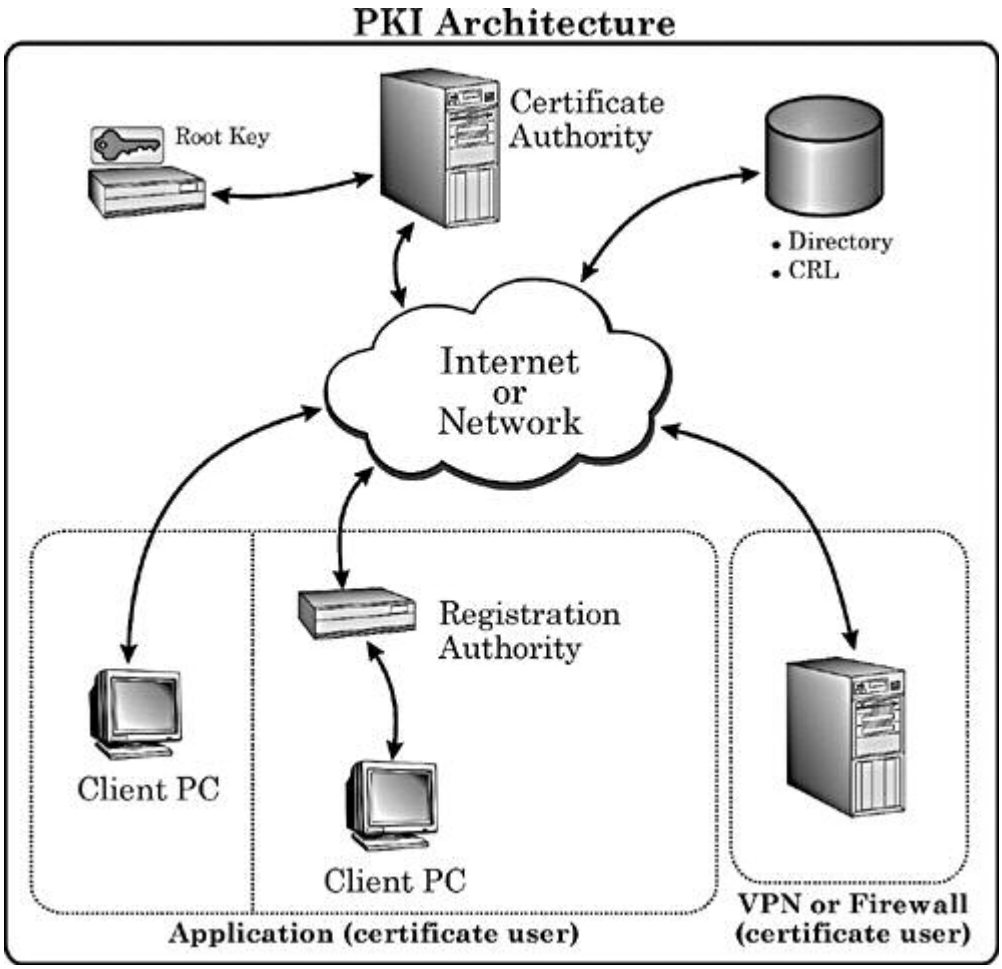


Fig 2. Structure of a Public Key Infrastructure or PKI.

Essential is that the problem is solved here in a hierarchical way. The CA for a user of Company A might be owned by this company. But when I send something to a user of company B what reason has he to trust the CA of my company. Therefore my CA has also a certificate that is signed this time by a CA but one “higher” in the CA-hierarchy (e.g. a government CA). Inthis way it is not one certificate that is recieved together with a signature but a list of (references to) certificates.

A web of trust

A second method for giving confidence that a public key really belongs to the person it is assumed to belong to is by using a *web of trust*. In a web of trust there are *keyservers*. Person C knows person D personally and knows he is a trustworthy person. Then person C puts a key of person D signed with his private key in the keyserver. Person B knows C and puts the key of C signed by him in the keyserver. PersonA recieves a message from D. Can he trust it? His computer sees that A trusts B, that B trusts C and C trusts D. The policy rules tell the computer that this

level of indirection is acceptable. The GUI of A gives a message: the message from D is trustworthy, but asks a confirmation from the user. As the user A knows personally C he accepts. This is a decentralised system where trust is defined by a policy database with facts and rules and where a decision can be done automatically (or partially automatically) or a human intervention may be needed (or only for some cases).

Fig. 3 illustrates the connection between trust and proof. Tiina claims access rights to the W3C. She adds to her claim the proof. The W3C can verify this by using the rules found on the site of Alan and the site of Kari.

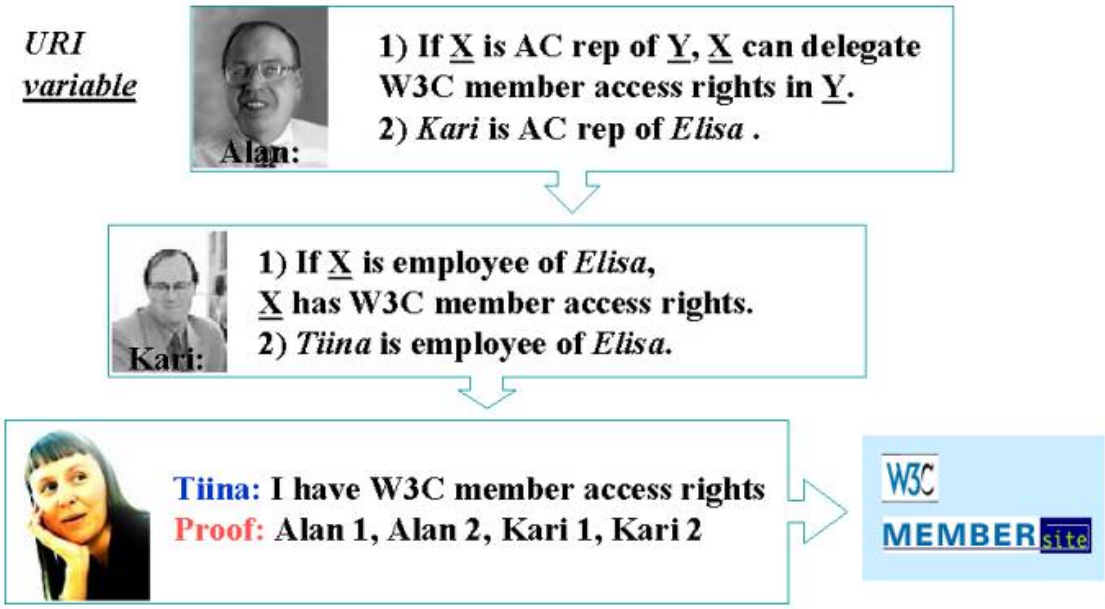


Fig. 3. Trust and proof. After Tim Berners-Lee.

The example in notation 3:
:Alan1 = {{:x w3c:AC_rep :y.} log:implies {:x w3c:can_delegate_access_rights :y.} ; log:forAll :x, :y.}
:Alan2 = {:Kari w3c:AC_REP :Elisa.}.
:Kari1 = {{:x DC:employee elisa:Elisa} log:implies {:x :has w3c:access_rights}; log:forAll :x.}.
:Kari2 = {:Tiina DC:employee elisa:Elisa.}.

{:proof owl:list :Alan1, :Alan2, :Kari1, Kari2} log:implies {:Tiina :has w3c:access_rights.}.

Tiina sends her proof rule together with :Alan1, :Alan2, :Kari1, :Kari2 to the W3C to claim her acess rights. However she adds also the following:
:Alan1 :verification w3c:w3c/acces_rights.

:Alan2 :verification elisa:Elisa/ac_rep.
:Kari1 :verification elisa:Elisa/Kari.
:Kari2 :verification elisa :Elisa/personnel.

These statements permit the w3c to make the necessary verifications.

The w3c has following meta-file (in sequence of execution):

{:proof owl:list :x} log:implies {:y :has w3c:access_rights.}; log:forAll :x, :y.
{:h owl:head :x. :h :verification :y. :t owl:tail :x. :proof owl:list :t.}log:implies {:proof owl:list :x};log:forAll :h, :x, :t, :y.
{:h :send_query :y} log:implies {:h :verification :y}; log:forAll :h, :y.

Of course :send_query is an action to be undertaken by the inference engine.
Does Tiina have to establish those triples herself? Of course not. She logs in to the w3c-site. From the site she recieves a N3-program that contains instructions (following the N3 presentation API; still to invent) for establishing a GUI where she enters the necessary data and the w3c-program then sends the necessary triples to the w3c. In a real environment the whole transaction will be further complicated by signatures and authentications i.e. security features.

There is no claim to executability of this piece of N3; neither of existence of the namespaces used.
This is a simple example but in practice much more complex situations could arise:

Joe recieves an executable in his mail. His policy is the following:
If the executable is signed with the company certificate then it is acceptable.
If the excutable is signed by Joe accept it.
If it comes from company X and is signed ask the user.
If the executable is signed, query the company CA server for acceptance.
If the CA server says no or don’t know reject the excutable.
If it is not signed but is from Joe accept.
If it is a java applet ask the user.
If it is active x it must be signed by Verisign.
In other cases reject it.

This gives some taste. A security policy can become very complicated.
OK, but why should RDF be used? If things happen on the internet it is

necessary to work with namespaces, URI's, URL's and , last nut not least, standards.

XML and namespaces

XML (Extensible Markup Language) is a subset of SGML (Standard General Markup Language). In its original signification 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. 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 quit interesting datastructures can be build (an example are the datastructures used in the modules Load.hs and N3Engine.hs from this thesis). 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 simpleness and the ease with which parsers and other tools can be build.
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. This could seem to be simpler but in fact it is more complex as now not only tags have to be treated e.g. by a parser but also attributes. The choice whether tags are used or attributes is dependent on personal taste and the application that is implemented with XML. Rules might be possible; one rule is: avoid attributes as they complicate the structure and make the automatical interpretation less easy. A question is also: do attributes add any

semantic information? It might be but it should then be made clear what the difference really is.
When there is no content or not any lower tags an abbreviation is possible:

<book title=”The semantic web” author=”Tim Berners-Lee”/>

where the closing tag is replaced by a single “/”.

An important characteristic of XML is the readability. OK it’s not like your favorite newsmagazine but for something which must be readable and handable for a computer it’s not that bad; it could have been hexadecimal code.
Though in the beginning XML was intended to be used as a vehicle of information on the internet it can be very well used in stand-alone applications too e.g. as the internal hierarchical tree-structure of a computer program . A huge advantage of using XML is the fact that it is standardized which means a lot of tools are available but which also means that a lot of people and programs can deal with it.
Very important is the hierarchical nature of XML. Expressing hierarchical data in XML is very easy and natural. This makes it a useful tool wherever hierarchical data are treated , including all applications using trees. XML could be a standard way to work with trees.

XML is not a language but a meta-language i.e. a language with as purpose to make other languages (“markup” languages).
Everybody can make his own language using XML. A person doing this only has to follow the syntaxis of XML i.e. produce wellformed XML. However (see further) more constraints can be added to an XML-language by using DTD’s and XML-schema, thus producing valid XML-documents. A valid XML-document is one that is in accord with the constraints of a DTD or XML-schema. To restate: an XML-language is a language that follows XML-syntax and XML-semantics. The XML-language can be defined using DTD’s or XML-schema.
If everybody creates his own language then the “tower-of-Babylon”-syndrom 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 fysiology of birds described however in another namespace:

```
<fysiology:temperature xmlns:fysiology=” http://fysiology.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>
<fysiology:temperature xmlns:fysiology=” http://fysiology.com/xml/”>
        43
</fysiology:temperature>
```

The version statement refers to the used version of XML (always the same).
XML gives thus the possibiliy of using more than one language in one object. What can a computer do with this? It can check the well-formedness of the XML-object. Then is a DTD or an XML-schema describing a language is available it can check the validity of the use of this language within the XML object. It cannot interprete the meaning of this XML-object at least not without extra programming. Someone can write a program (e.g. a veterinary program) that makes an alarm bell sound when the temperature of a certain bird is 45 and research on the site “http://fysiology.com/ “ has indicated a temperature of 43 degrees Celsius.

<u>Semantics of XML</u>

The main “atoms” in XML are tags and attributes. Given the interpretation function for tags and attributes and a domain if t1 is a tag then I(t1) is supposed to be known. If a1 is an attribute then I(a1) is supposed to be known. If c1 is content then I(c) is supposed to be known. Given the structure:
x = <t1><t2>c</t2></t1>
I(x) could be : I(t1) and I(t2) and I(c). However here the hierarchical structure is lost. A possibility might be: I(x) = I(t1)[I(t2)[I(c)]] where the signs “[“ and “]” represent the hierarchical nature of the relationship.
It might be possible to reduce the semantics of XML to the semantics of RDF by declaring:

t1 :has :a1. t1 :has :c1. t1 :has t. where t1 is a tag, a1 is an attribute, c1 is content and t is an XML-tree. The meaning of :has is in the URI where :has refers to. Then the interpretation is the same as defined in the semantics of RDF.
The text above is about well-formed XML. DTD's and XML-schema change the semantic context as they give more constraints that restrict the semantic interpretation of an XML-document. When an XML- document conforms to a DTD or XML-schema it is called a valid XML-document.

DTD and XML-Schema

These two subjects are not between the main subjects relevant for this thesis, but it are important tools that can play a role in the Semantic Web so I will discuss a small example. Take the following XML-object:

```
<?xml version="1.0.1" ?>
<!DOCTYPE bird SYSTEM "http://www.bird.com/bird.dtd">
<bird frequency = "2">
  <wing>
    large
  </wing>
  <color>
    yellow
  </color>
</bird>
```

The DOCTYPE line indicates the location of the DTD that describes the XML-object. (Supposedly bird-watchers are indicating the frequency with which a bird has been seen, hence the attribute frequency).

And here is the corresponding DTD (the numbers are not part of the DTD but added for convenience of the discussion):

```
1) <!DOCTYPE bird
2)   [<!ELEMENT bird (wing, color?, place+)>
3)      <!ATTLIST bird frequency CDATA #REQUIRED>
4)      <!ELEMENT wing PCDATA>
5)      <!ELEMENT color PCDATA>
6)      <!ELEMENT place PCDATA>
7)   ]>
```

Line 1 gives the name (which is the root element of the XML-object) of the DTD corresponding to the DOCTYPE declaration in the XML-object. In line 2 the ELEMENT (= tag) bird is declared with the

indication that there are three elements lower in the hierarchy. The element wing may only occur once in the tree beneath bird; the element color may occur 0 or 1 times (indicated by the “?”) and the element place may occur one or more times (indicated by “+”). An * would indicate 0 or more times.
In line 3 the attributes of the element bird are defined. There is only one attribute “frequency”. It is declared of being of type CDATA (= alphanumerical) en #REQUIRED which means it is obligatory.
In lines 4, 5 and 6 the elements wing, color and place are declared as being of type PCDATA (= alphanumerical). The diference between CDATA and PCDATA is that PCDATA will be parsed by the parser (e.g. internal tags will be recognized) and CDATA will not be parsed.

DTD has as a huge advantage it ease of use. But there are a lot of disadvantages.
[http://pro.html.it/print_articolo.asp?id=175].

1) a DTD object is not in XML syntaxis. This creates extra complexity and also needless as it could have been easily defined in XML-syntaxis.
2) The content of tags is always #PCDATA = alphanumerical; the possibility to define and validate other types of data (like e.g. numbers) is not possible.
3) There is only one DTD-object; it is not possible to import other definitions.

To counter the critics on DTD W3C devised XML-Schema. XML-Schema offers a lot more possibilities for making definitions and restrictions as DTD but at the price of being a lot more complex. (Note: again the line numbers are added for convienence).[http://www.w3schools.com/schema/schema_schema.asp].

```
1)  <xml version="1.0"?>
2)  <xs:schema xmlns:xs=“http://www.w3.org/2001/XMLSchema
3)                xs:targetNameSpace="http://www.test/#bird">
4)  <xs:element name=“bird“>
5)    <xs:complexType>
6)      <xs:sequence>
7)        <xs:attribute name=”frequency” type=”xs:integer”/>
8)        <xs :element name=“wing” type=”xs:string/ maxOccurs=1
    minOccurs=1/>
9)        <xs :element name=“color” type=”xs:string maxOccurs=1
    minOccurs=0/>
10)       <xs :element name=“place” type=”xs:string minOccurs=1/>
11)     </xs :sequence>
```

12) </xs:complexType>
13) </xs :element>
14) </xs :schema>

Line 1 : an XML-Schema is an XML-object. The root of an XML_Schema is always a schema tag. It can contain attributes, here the namespace where XML-Schema is defined and the location of this schema definition.
In the XML-object bird the statement:
<xs:schemaLocation="http://www.test/bird.xsd">
can indicate the location of the XML-Schema.
In line 2 the namespace of XML-Schema is defined (there you can find all the official documents).
Line 3 defines what the target namespace is i.e. to which namespace the elements of the XML-object bird belong that do not have a namespace prefix.
Line 4 defines the root element bird of the defined XML-object. (The root of the schema document is <xs:schema …>).
Line 5: bird is a complex element. Elements that have an element lower in the hierarchy or/and an attribute are complex elements. This is declared with xs:complexType.
Line 6: complex types can be a sequence, an alternative or a group.
Line 7: the definition of the attribute frequency. It is defined as an integer (this was not possible with a DTD).
Line 8: the defintion of the element "wing". This element can only occur one time as defined by the attributes maxOccurs and minOccurs of the element xs:element.
Line 9: the element "color" can occur 0 or 1 times.
Line10: the element "place" can oocur 1 or more times.
Line 11, 12, 13, 14: closing tags.

Because the syntaxis of XML-Schema is XML it is possible to use elements of XML-Schema in RDF(see further) e.g. for defining integers.

<u>Other internet tools</u>

For completeness some other W3C tools are mentionned for their relevance in the Semantic Web (but not for this thesis):

1) XSL.[W3SCHOOLS]
   XSL consists of three parts:
   a) XSLT (a language for transforming XML documents).
      Instead of the modules N3Parser en Load who transform

Notation 3 to an XML-object, it is possible to transform Notation 3 to RDF (by one of the available programs), then apply XSLT for transforming the RDF-object into the desired XML-format.

b) XPath (a language for defining parts of an XML document).
c) XSL Formatting Objects (a vocabulary for formatting XML documents).

2) SOAP[W3SCHOOLS]:
A SOAP message is an XML-object consisting of a SOAP-header who is optional, a SOAP-envelope that defines the content of the message and a SOAP-body that contains the call and response data. The call-data have as a consequence the excution of a remote procedure by a server and the response data are sent from the server to the client. SOAP is an important part of Web Services.

3)WSDL[W3SCHOOLS] and UDDI: WSDL stand for: Web Services Description Language. A WSDL-description is an XML-object that describes a WebService. Another element of Web Services is UDDI (Universal Description, Discovery and Integration service). UDDI is the description of a service that should permit finding web-services on the internet. It is to be compared with Yellow and White Pages for telephony.

URI's and URL's

What is a URI? URI means Uniform Resource Indicator.
The following examples illustrate URI that are in common use. [http://www.isi.edu/in-notes/rfc2396.txt].

ftp://ftp.is.co.za/rfc/rfc1808.txt
-- ftp scheme for File Transfer Protocol services

gopher://spinaltap.micro.umn.edu/00/Weather/California/Los%20Angeles
-- gopher scheme for Gopher and Gopher+ Protocol services

http://www.math.uio.no/faq/compression-faq/part1.html
-- http scheme for Hypertext Transfer Protocol services

mailto:mduerst@ifi.unizh.ch
-- mailto scheme for electronic mail addresses

news:comp.infosystems.www.servers.unix
-- news scheme for USENET news groups and articles

telnet://melvyl.ucop.edu/
-- telnet scheme for interactive services via the TELNET Protocol

URL stands for Uniform Resource Locator. This is a subset of URI. An 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 Uniform Resource Name.

In this thesis only URL's will be used and only http as protocol.
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 searchpath 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>

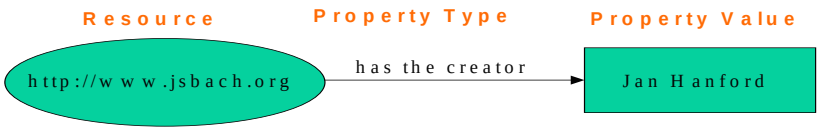
A named anchor thus indicates a location within a document. The named anchor can be called e.g. by:

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

Resource Description Framework RDF

[RDF Primer]

RDF is a language. The semantics are defined by [RDF_SEMANTICS]; three syntaxes are known: XML-syntax, Notation 3 and N-triples. N-triples is a subset of Notation 3 and thus of RDF.
Very basically RDF consist of triples: subject - predicate - object. This simple statement however is not the whole story; nevertheless it is a good point to start.
An example from [www.albany.edu/~gilmr/metadata/rdf.ppt]:
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 ... gives a graphical view of this.



In simplified RDF this becomes:

```
<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>
```

xmlns stands for: XML Namespace. The first namespace refers to the document describing the (XML-)syntax of RDF; the second namespace 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.
There is also an abbreviated rdf-syntax. The example above the 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”>
  </rdf:Description>
</rdf:RDF>
```

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>
```

or in abbreviated form:

```
<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:habitat=“forest”>
  </rdf:Description>
</rdf:RDF>
```

Other abbreviated forms exists but this is out of scope for this thesis.

The container model of RDF:

Three container types exist in RDF:

1) a bag: an unordered list of resources or literals. Duplicates are permitted.
2) a sequence: an ordered list of resources or literals. Duplicates are permitted.
3) An alternative: a list of resources or literals that represent *alternative* values for a predicate.

Here is an example of a bag. For a sequence use *rdf*:seq and for an alternative use *rdf*:alt.

```
<rdf:RDF>
  <rdf:Description about="http://www.birds.com/birds/colors/">
    <bi:colors>
   <rdf:Bag ID="bird_colors">
          <rdf:li resource="http://www.birds.com/birds/colors#yellow"/>
          <rdf:li resource="http://www.birds.com/birds/colors#red"/>
          <rdf:li resource="http://www.birds.com/birds/colors#green"/>
      </rdf:Bag>
    </bi:colors>
  </rdf:Description>
</rdf:RDF>
```

Note that the “bag” statement has an id which makes it possible to refer to the bag.

```
<rdf:Description about="#bird_colors">
  <dc:Creator>Guido Naudts</dc:Creator>
</rdf:Description>
```

It is also possible to refer to all elements of the bag at the same time with the “aboutEach” attribute.

```
<rdf:Description aboutEach="#bird_colors">
  <bi:Description>See bird manual</bi:Creator>
</rdf:Description>
```

This says that a description of each color can be found in the manual.

Reification

Reification means describing a RDF statement by describing its separate elements. E.g. following example:

```
<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”>
  </rdf:Description>
</rdf:RDF>
```

becomes:

```
<rdf:RDF xmlns:rdf=”http://www.w3.org/1999/02/22-rdf-syntax-ns#”
xmlns:dc=”http://purl.org/DC/”>
  <rdf:Description>
    <rdf:subject resource="http://www.jsbach.org/"/>
    <rdf:predicate resource="http://purl.org/DC/Creator" />
    <rdf:object>Jan Hanford</rdf:object>
    <rdf:type resource="http://www.w3.org/1999/02/22-rdf-syntax-
ns#Statement" />
  </rdf:Description>
</rdf:RDF>
```

RDF data model

Sets in the model :
1) There is a set called Resources.
2) There is a set called Literals.
3) There is a subset of Resources called Properties.
4) There is a set called Statements, each element of which is a triple of the form
{pred, sub, obj}
where pred is a property (member of Properties), sub is a resource (member of Resources), and obj is either a resource or a literal (member of Literals).

RDF:type is a member of Properties.
RDF:Statement is a member of resources but not contained in Properties.
RDF:subject, RDF:predicate and RDF:object are in Properties.

Reification of a triple {pred, sub, obj} of Statements is an element r of Resources representing the reified triple and the elements s_1, s_2, s_3, and s_4 of Statements such that
s_1: {RDF:predicate, r, pred}
s_2: {RDF:subject, r, subj}
s_3: {RDF:object, r, obj}
s_4: {RDF:type, r, [RDF:Statement]}

s1 means that the predicate of the reified triple r is pred. The type of r is RDF:Statement.

RDF:Resource indicates a resource.
RDF:Property indicates a property. A property in rdf is a first class object and not an attribute of a class as in other models. A property is also a resource.

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. Notwithstanding its simple structure a great deal of information can already be expressed with it. One of the strong points of RDF lies in its simplicity with as a consequence that reasoning engines can be constructed in a fairly simple way thanks to easy manipulation of data structures and simple unification algorithms.

Notation 3

Here is an explanation of the points about Notation 3 or N3 that were used in this thesis. This language was developed by Tim Berners-Lee and Dan Connolly and represents a more human manipulable form of the RDF-syntax with in principle the same semantics. For somewhat more information see : [RDF Primer].

First some basic notions about URI's : URI means Universal Resource Indicator. In this thesis only URI's that are URL's are used. URL means Universal Resource Locator. URL's are composed of a protocol indicator like http and file (what are the most commonly used), a location indication like www.yahoo.com and eventually a local resource indicator like #subparagraph giving e.g . http://www.yahoo.com#subParagraph. See also : http://www.w3.org/Adressing/ .

In N3 URI's can be indicated in a variety of different ways :

- <http://www.w3.org/2000/10/swap/log#log:forAll> : this is the complete form. The namespace is in its complete form. The N3Parser (see further) always generates first the abbreviated form as used in the source ; this is followed by the complete URI.
- <#param> : the complete form is : <URL_of_current_document#param>.
- <> : the URI of the current document.
- **:xxx** : This is the use of a prefix. A prefix is define in N3 by the @prefix instruction :
  @prefix ont: <http://www.daml.org/2001/03/daml-ont#>.
  This defines the prefix ont: . Note the finishing point in the @prefix instruction.
  So ont:TransitiveProperty is in full form <http://www.daml.org/2001/03/daml-ont#TransitiveProperty> .

- **:** : a single double point 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 **: <#> .**

Basically Notation 3 works with « triples » who have the form : <subject> <verb> <object> where subject, verb and object are atoms. An atom can be either a URI (or a URI abbreviation) or a variable. But some more complex structures are possible and there also is some " syntactic sugar". Verb and object are also called property and value which is anyhow the semantical meaning.

Two substantial abbreviations 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.
These properties are separated by a point-comma.
A verb or property can have several values e.g.
:bird :color :blue, yellow, black.
This means that the bird has 3 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 signs '[' and ']' are used for this feature. [:can :swim]. means there exists an anonymous subject x that can swim ; e.g. I have seen a bird but I do not know which bird. The abbreviations for propertylist and objectlist can here be used too :

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

Some more syntactic sugar must be mentioned.

:lion :characteristics :mammal.

can be replaced by:

:lion has :characteristics of :mammals.

The words "has" and "of" are just eliminated by the parser.

:lion :characteristics :mammals.

can be replaced by:

:mammals is :characteristics of :lion.

Again the words “is” and “of” are just eliminated; however in this case subject and object have to be interchanged.

The property rdf:type can be abbreviated as “a”:

:lion a :mammal.

really means:

:lion rdf:type :mammal.

The property owl:equivalentTo can be abbreviated as “=”, e.g.

:daml:EquivalentTo = owl:equivalentTo.

meaning the semantic equivalence of two notions or things.

This notion of equality probably will become very important in future for assuring interoperability between different systems on the internet: if A uses term A meaning the same as term B used by B, this does not matter if this equivalence can be expressed and found.

The logic layer

In http://www/w3.org/200/10/swap/log# an experimental logic layer is defined for he semantic web. An overview of the most salient features (the N3Engine only uses log:implies, log:forAll, log:forSome and log:Truth):

log:implies : this is the implication.

{:rat a :rodentia. :rodentia a :mammal.} log:implies {:rat a :mammal}.

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

this log:forAll :a, :b, :c.

indicates that :a, :b and :c are universal variables.

The word “this” stands for the scope enveloping the formula. In the form above this is the whole document. When between bracelets it is the local scope: see [PRIMER]. In this thesis this is not used.

log:forSome does the same for existential variables.

This log:forSome :a, :b, :c.

log:Truth : states that this is a universal truth. This is not interpreted by the N3Engine.

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.

Semantics of N3

The semantics of N3 are the same of the semantics of RDF. See [RDFM] which gives a model-theoretic semantics for 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

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.

Informally this means that every URI represent 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 containing 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.

As an example take the triple:

:bird :color :yellow.

In the set of URI's there will be things like: :bird, :mammal, :color, :weight, :yellow, :blue etc...

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).

RDF Schema

Withe RDF Schema comes the possibility to use constraints i.e. limiting the values that can be an element of defined sets. Say "rats" is a set and it is expressed that "rats" is a subclass of "mammals". This is a restriction on the set "rats" as this set can now only contain elements that are "mammals" and thus have all properties of mammals.
Here follows an overview of the important concepts. The first-order descriptions are taken from:
[Champin] and put in SWeLL format.
The RDF Schema namespace is indicated with rdfs.

rdfs:subPropertyOf : A property is a relation between sets and consists of a set of tuples. A subproperty is a subset of this set.
Rule: {{ :s :p1 :o. :p1 rdfs:subPropertyOf :p2. } log:implies { :s :p2 :o}} a log:Truth; log:forAll :s,:p1,:o,:p2.
Since subPropertyOf defines a subset, transitivity holds:
rfds:subPropertyOf a owl:TransitiveProperty. with the definition of owl:TransitiveProperty:
{{:p a owl:TransitiveProperty. :a :p :b. :b :p :c.} log:implies {:a :p :c}} a log:Truth; log:forAll :a, :b, :c, :p.
Cycles are not permitted. Cycles have as a consequence that a subproperty is its own subproperty. This can be expressed as:

{:p rdfs:subPropertyOf :p} a log:FalseHood; log:forAll :p.
Also:
{{:p a rdfs:subPropertyOf} log:implies {:p a rdf:property}} a log:Truth; log:forAll :p.

rdfs:Class: a class defines semantically a set of URI's. The set is defined by indicating one way or another which items are in the class.

rdfs:subClassOf:
The meaning of subClassOf is analogous to subpropertyOf:
{{ :s :p1 :o. :p1 rdfs:subClassOf :p2. } log:implies { :s :p2 :o}} a log:Truth; log:forAll :s,:p1,:o,:p2.

And of course:
rdfs:subClassOf a owl:TransistiveProperty.

Every class is a subclass of rdf:Resource:
{{:c a rdfs:Class.} log:implies {:c rdfs:subClassOf rdf:Resource}} a log:Truth;log:forAll :c.
rdf:Resource a rdfs:Class.

rdfs:domain and rdfs:range:

First:
rdfs:domain a rdf:property.
rdfs:range a rdf:property.

The domain(s) of a property defines which individuals can have the property i.e. the class(es) to which those individuals belong. A property can have more than one domain. The range of a property defines to which class the values of the property must belong. A property can have only one range:

{:p rdfs:range :r1. :p rdfs:range :r2. :r1 owl:differentIndividualFrom :r2} a log:FalseHood; log:forAll :p, :r1, :r2.

When at least one domain is defined the subject of a property must belong to some domain of the property. When a range is defined the object of a property must belong to the defined range of the property:

{{:s :p :o. :p rdfs:domain :d.} log:implies {:s rdf:type :d1. :p rdfs:domain :d1}} a log:Truth; log:forAll :s, :p, :o; log:forSome :d, :d1.

This rule can not be handled by the engine proposed in this thesis as it has a multiple consequent. However the rule can be put as follows:

{{:s :p :o. :p rdfs:domain :d. :s rdf:type :d1} log:implies { :p rdfs:domain :d1}} a log:Truth; log:forAll :s, :p, :o; log:forSome :d, :d1.

The rule for the range is simpler:
{{:s :p :o. :p rdfs:range :d.} log:implies {:o rdf:type :d}} a log:Truth; log:forAll :s, :p, :o, :d.

rdfs:Literal denotes the set of literals.
rdfs:Literal a rdfs:Class.

rdfs:Container: has three subclasses: rdf:Bag, rdf:Seq, rdf:Alt.
rdf:Bag rdfs:subClassOf rdfs:Container.
rdf:Seq rdfs:subClassOf rdfs:Container.
rdf:Alt rdfs:subClassOf rdfs:Container.

Members of a container are modlled by:
rdf:_1, rdf:_2, etc...
These are properties (rdf:_1 a rdf:Property.) and are instance of rdfs:ContainerMembershipProperty so:
rdf:_1 a rdfs:ContainerMembershipProperty.
rdfs:ContainerMembershipProperty rdfs:subClassOf rdf:Property.

rdfs:ConstraintResource and rdfs:ConstraintProperty:

Some definitions:
rdfs:ConstraintResource rdfs:subClassOf rdf:Resource.
rdfs:ConstraintProperty rdfs:subClassOf rdf:Property.
rdfs:ConstraintProperty rdfs:subClassOf rdfs:ConstraintResource.
rdfs:range a rdfs:ConstraintProperty.
rdfs:domain a rdfs:ConstraintProperty.

The use of these two classes is not very clear.

rdfs:seeAlso and rdfs:isDefinedBy:

rdfs:seeAlso points to alternative descriptions of the subejct resource e.g.
:birds rdfs:seeAlso <http://www.americanBirds.com/>.
rdfs:isDefinedBy is a subproperty of rdfs:seeAlso and points to an original or authoritative description.

rdfs:seeAlso a rdf:Property.

rdfs:isDefinedBy rdfs:subPropertyOf rdfs:seeAlso.

rdfs:label and rdfs:comment:

The purpose of rdfs:label is to give a “name” to a resource e.g.

rdf:Property rdfs:label “An rdf property.”

rdfs:comment serves for somewhat longer texts.

Ontology Web Language (OWL)

Here is a list of ontology elements that are part of OWL:

> rdf:type, rdf:Property, rdfs:subClassOf, rdfs:subPropertyOf, rdfs:domain, rdfs:range, owl:Class, owl:sameClassAs, owl:DisjointWith, owl:oneOf, owl:unionOf, owl:intersectionOf, owl:complementOf, owl:samePropertyAs, owl:inverseOf, owl:DatatypeProperty, owl:ObjectProperty, owl:SymmetricProperty, owl:UniqueProperty, owl:UnambiguousProperty, owl:TransitiveProperty, owl:Restriction, owl:onProperty, owl:toClass, owl:hasClass, owl:hasValue, owl:minCardinality, owl:maxCardinality, owl:cardinality, owl:sameIndividualAs, owl:differentIndividualFrom, owl:List, owl:first, owl:rest, owl:nil.

The rdf and rdfs elements have already been discussed.

There are two main parts to OWL:

- the definition of datatypes based on XML Schema. Datatypes are elements of owl:Datatype.
- The object domain: the description of object classes into classes. Classes are elements of owl:Class. This gives the first statement:

  owl:Class rdfs:subClassOf rdfs:Class.

Two class names are already predefined, namely the classes owl:Thing and owl:Nothing. Every object is a member of owl:Thing, and no object is a member owl:Nothing. Consequently, every class is a subclass of owl:Thing and owl:Nothing is a subclass of every class.

This gives two rules :

{{:p a owl:Class} log:implies {:p rdfs:subClassOf owl:Thing}} a log:Truth; log:forAll :p.
{{:p a owl:Class} log:implies {owl:Nothing rdfs:subClassOf :p}} a log:Truth; log:forAll :p.

OWL Lite is a subset of OWL. The following discussion will mostly be about OWL Lite.

OWL Lite Equality and Inequality

owl:sameClassAs: expresses equality between classes e.g. :mammals owl:sameClassAs :mammalia.

owl:sameClassAs rdfs:subPropertyOf rdfs:subClassOf.
{{:c1 owl:sameClassAs :c2. :i1 a :c1.} log:implies {:i1 a :c2}} a log:Truth; log:forAll :c1, :c2, :i1.

owl:samePropertyAs: expresses the equality of two properties e.g. bi:tall owl:samePropertyAs ma:huge. when two ontologies use a different term with the same semantics.

{{:p1 owl:samePropertyAs :p2. :s :p1 :o.} log:implies { :s :p2 :o}} a log:Truth; log:forAll :p1, :p2, :s, :o.

owl:sameIndividualAs: expresses the equality of two individuals e.g. ma:lion1 owl:sameIndividualAs zo:leeuw_zoo.

Two rules are the consequence of this property:
{{:s1:p :o. :s2 owl:sameIndividualAs :s1} log:implies {:s2 :p :o1}} a log:Truth; log:forAll :o, :p, :s1, :s2.
{{:s :p :o1. :o2 owl:sameIndividualAs :o1} log:implies {:s :p :o1}} a log:Truth; log:forAll :s, :p, :o1, :o2.

owl:differentIndividualFrom: states that two individuals are not equal e.g. :mammals owl:differentIndividualFrom :fishes. How to put this in a rule?
Or said otherwise: if the engine knows :a owl:differentIndividualFrom :b, what can it deduce? When the statement :a sameIndividualAs :b also exist then there is of course a contradiction. This could be used as a fact matching with a goal produced by a rule.

OWL Lite property characteristics:

owl:inverseOf: one property is the inverse of another property e.g. hasChild is the inverse of hasParent. {:a :hasChild :b} log:implies {:b :hasParent :a}.
{{:p1 owl:inversOf :p2. :s :p1 :o.} log:implies {:o :p2 :s}} a log:Truth; log:forAll :p1, :p2, :s, :o.

owl:TransitiveProperty: properties can be transitive e.g. smaller than ...
Rule:
{{:p a owl:TransitiveProperty. :a :p :b. :b :p :c.} log:implies {:a :p :c}} a log:Truth; log:forAll :a, :b, :c, :p.
Example of a transitive property:
rdfs:subClassOf a owl:TransitiveProperty.

owl:SymmetricProperty: properties can be symmetric e.g.{ :a :friend :b } log:implies {:b :friend :a}.
{{:p a owl:SymmetricProperty. :a :p :b. } log:implies {:b :p :a}} a log:Truth; log:forAll :a, :b, :p.

owl:FunctionalProperty: this is a property that has 0 or 1 values e.g. :animal1 :hasFather :animal2. (Not all animals do have a father but if they do there is only one.)
{{:p a owl:FunctionalProperty. :s :p :o1. :s :p :o2. } log:implies{:o1 owl:sameIndividualAs :o2}} a log:Truth; log:forAll :p, :s, :o1, :o2.

owl:InverseFunctionalProperty: also called an unambigous property. :animal1 :isFatherOf :animal2.
{{:p a owl:FunctionalProperty. :s1 :p :o. :s2 :p :o. } log:implies{:s1 owl:sameIndividualAs :s2}} a log:Truth; log:forAll :p, :s1, :s2, :o.

Property restrictions:

allValuesFrom: this is a restriction on the values of the object that go with a duo (subject, property). The interpretation followed here is: when a subject s belonging to a certain class S has the property p with restriction to class O then the relation: s p o must be valid where o is an instance of O. Here is an example in RDF from [http://www.daml.org/2002/06/webont/owl-ex]:

```
<owl:Class rdf:ID="Person">
  <rdfs:subClassOf rdf:resource="#Animal"/>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasParent"/>
      <owl:allValuesFrom rdf:resource="#Person"/>
```

```
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction owl:cardinality="1">
      <owl:onProperty rdf:resource="#hasFather"/>
    </owl:Restriction>
  </rdfs:subClassOf>
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#shoesize"/>
      <owl:minCardinality>1</owl:minCardinality>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This means that : a person is an animal ; if a person has a parent then he is a person; a person has only one father; his shoesize is minimally 1.

It is interesting to put this in N3.

```
{<#Person> rdfs:subClassOf  <#Animal>;
        rdfs:subClassOf
            {owl:Restriction owl:onProperty <#hasParent>;
                        owl:allValuesFrom <#Person>};
        rdfs:subClassOf
            {owl:Restriction owl:cardinality "1";
                        owl:onProperty <#hasFather>};
        rdfs:subClassOf
            {owl:Restriction owl:onProperty <#shoeSize>;
                        owl:minCardinality "1"}}.
```

Three intertwined triples are necessary for using the notion “allValuesFrom”.
A rule:
{{:c a {owl:Restriction owl:onProperty :p1; owl:allValuesFrom :o1. :s1 owl:Class :c}. :s1 :p1 :o2 } log:implies {:o2 a :o1}} a log:Truth; log:forAll :s1, :p1, :o1, :o2, :c.
Add the facts:
:a <#hasParent> :b.
:a owl:Class :c.
:c a {owl:Restriction owl:onProperty <#hasParent>; owl:allValuesFrom <#Person>}.
and put the query:
_:who a <#Person>.

with the answer:
:b a <#Person>.

someValuesFrom: this is a restriction on the values of the object that go with a duo (subject, property). The interpretation followed here is: when a subject s belonging to a certain class S has the property p with restriction to class O then the relation: s p o must be valid where o is an instance of O at least for one instance o. Contrary to allValuesFrom only some values (at least one) of the class do need to belong to the restriction.Here is an example in RDF:

```
<owl:Class rdf:ID="#Person">
  <rdfs:subClassOf>
    <owl:Restriction>
      <owl:onProperty rdf:resource="#hasClothes"/>
      <owl:someValuesFrom rdf:resource="#Trousers"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

This means that : “Person” is a class with property “hasClothes” and at least one value of “hasClothes” is “trousers”.

It is interesting to put this in N3.

```
{<#Person> rdfs:subClassOf
        {owl:Restriction owl:onProperty <#hasClothes>;
                        owl:somelValuesFrom <#Trousers>}.
```

Three intertwined triples are again necessary for using the notion “someValuesFrom”.
A rule:
{{:c a {owl:Restriction owl:onProperty :p1; owl:someValuesFrom :o1}.
:s1 owl:Class :c.:s1 :p1 :o2 } log:implies {:o2 a :o1}} a log:Truth;
log:forAll :s1, :p1, :o1; log:forSome :o2.

The only difference here in the rule compared with the rule for allValuesFrom is in “log:forSome :o2”.
Add the facts:
:a <#hasClothes> :b.
:a owl:Class :c.
:c a {owl:Restriction owl:onProperty <#hasClothes>;
owl:someValuesFrom <#Trousers>}.
and put the query:

_:who a <#Trousers>.
Here the rule means: if there is a triple {:a <#hasClothes> :b} with :a belonging to class :c, there should be at least one triple {:a <#hasClothes :t} where :t is a “Trousers”. The above query then does not make much sense; the someValuesFrom is a restriction on the content of a database and should be enforced before queries are made (see further).

OWL Lite restricted cardinality:

minCardinality: this is stated on a property with respect to a particular class. In OWL Lite only 0 and 1 are permitted as values. In the full version there are no limitations on cardinality. If the the property habitat has minCardinality 1 for the class Animal, this means that each animal should have at least one habitat (which seems reasonable).

```
<owl:Class rdf:ID="Animal">
  <rdfs:subClassOf>
    <owl:Restriction owl:minCardinality="1">
      <owl:onProperty rdf:resource="#habitat"/>
    </owl:Restriction>
  </rdfs:subClassOf>
</owl:Class>
```

<#Animal> rdfs:subClassOf { owl:Restriction owl:minCardinality “1”; owl:onProperty <#habitat>}.

A first trial for a rule:
{{:s1 owl:Class :c. :c a {owl:Restriction owl:minCardinality :n; owl:onProperty
:p.}. :n :greater “0”. } log:implies { :s1 :p :o.}} a log:Truth; log:forAll :s1, :c, :n, :p; log:forSome :o.

This rule however only states that at least one value of the property exist while, if minCardinality e.g. is 3, 3 values should exist. It seems necessary to describe actions or orders e.g. an order saying count the occurences of the values for that property and check if the minCardinality value is satisfied. This could be something like:
:c :cardinality {:a :p :o} ; log:forSome :o.
:c :greater :min. where :min is the minimum cardinality of the investigated property giving the rule:
{{:s1 owl:Class :c. :c a {owl:Restriction owl:minCardinality :n; owl:onProperty

:p.} :n1 :cardinality {:s1 :p :o}. } log:implies { :n1 :greater :n.}} a log:Truth; log:forAll :s1, :c, :n, :p, :n1, :o. where :cardinality and :greater have to be interpreted by the engine.

This means that some ontology proper of the engine is necessary and that this ontology must be interpreted by the engine. In fact to impose an ontology a metalanguage is needed i.e. a language which speaks about the language so that properties like cardinality etc... can be checked. This metalanguage can have a RDF-syntaxis but it seems that semantically some extensions are necessary e.g. like for counting cardinalities.

The meta-rules and facts for the engine have to be divide into 2 sets:

1) rules to be applied during the preparatory phase where restrictions are tested and inconsistencies detected; where types are assigned.
2) rules for use during the inference of an answer to a query.

To impose restrictions the engine must detect inconsistencies which can be done by a query:
_:what :inconsistentWith _:ontologyItem.
This query would then return the list of all inconsistencies (= all possible solutions to the query)).
This is illustrated by the following rule form [OWL Rules]):

{{:s a [a owl:Restriction; owl:onProperty :p; owl:minCardinality :min]. {:s :p :o} math:proofCount :n. :n math:lessThan :min} log:implies {:s :inconsistentWith owl:Restriction}} a log:Truth; log:forAll :s, :p, :min, :o, :n.

Some explanations are in order. math:proofCount would give the number of possible triples that {:s :p :o} that satisfy the current substitutions for these variables (there normally are substitutions for :s and :p based on the owl:Restriction; for :o anything can be substituted.) This number can be found by posing a query: {subst(:s) subst(:p) :o} where subst(:s) means the current substitution for :s.

This has an important consequence for the engine: it must be able to make subqueries and using the results of a subquery within an ongoing query. Thus the engine must be recusively callable.

The rules should take into account the different formats possible for stating restrictions with RDF. Implementing completely OWL Lite in this way is out of scope for this thesis and seems to be more the subject for a Ph.D. Implementing OWL Lite without a set of meta-rules; just by interpreting the N3 input and detecting all restrictions and applying them

is not so difficult. The main disadvantage however is that the result would be an engine that is very difficult to adapt to changes in the restrictions or for working with different sets of restrictions.

maxCardinality: for maxCardinality there is no restriction on the values in OWL Lite. The meaning is supposed to be clear.

cardinality: a certain value n for the cardinality of a property means that that property has the same value n for minCardinality and for maxCardinality.

OWL Lite datatypes :

There are in OWL two types of property : DatatypeProperty and ObjectProperty. An ObjectProperty relates an object to an object while a DatatypeProperty relates an object to a datatype value. The value of a datatype can be indicated with XML Schema.
An example of ObjectProperty:

owl:objectProperty <#wingColor>; rdfs:domain <#bird>; rdfs:range <#color>.

An example of DatatypeProperty:

owl:DatatypeProperty <#wingSize>; rdfs:comment
“wingsize is a DatatypeProperty whose range is xsd:decimal”;
rdfs:range <http://www.w3.org/2000/10/XMLSchema#decimal>.

OWL List primitives

The list primitives for owl are:
owl:List, owl:first, owl:rest, owl:nil.

Thise xamples creates the list (1,2,3):
:list a owl:List.
:list owl:first “1”; owl:rest [owl:first “2”; owl:rest [owl:first “3”; owl:rest owl:nil]].

Elements of full owl:

oneOf: or enumerated classes : a class is described by an enumeration.
Example: :owl_family owl:oneOf :barn_owl, :typical_owls.

Elements of sets: unionOf, complementOf, intersectionOf, disjointWith:

I will give only rules for two classes. For a list of classes more list primitives are necessary.

unionOf: the union of classes. Example:
:vertebrae owl:unionOf :pisces, :amphibia, :reptilia, :aves, :mammalia.

{{:c1 a owl:Class. :c2 a owl:Class. :c3 owl:unionOf :c1, :c2. :e a :c1.} log:implies {:e a :c3}} a log:Truth; log:forAll :c1, :c2, :c3.

complementOf: the class(es) that is (are) complementary to another class. Example: :negative_numbers owl:complementOf :positive_numbers. Understood had to be that there is a common superclass composed of the complementary classes:
:number owl:unionOf :negative_numbers, positive_numbers.

{{:c1 a owl:Class. :c2 a owl:Class. :e a :c1. :c1 owl:complementOf :c2.} log:implies {:e a :c2}} a log:FalseHood; log:forAll :c1, :c2, :e.

intersectionOf: a class is the intersection of two other classes. Example: :fish-eaters rdfs:subClassOf :birds. :predators rdfs:subClassOf :birds. :marine_predators owl:intersectionOf :fish_eaters, :predators.

{{:c1 a owl:Class. :c2 a owl:Class. :c owl:intersectionOf :c1, :c2. :e a :c1. :e a :c2.} log:implies {:e a :c}} a log:Truth; log:forAll :c1, :c2, :c, :e.

owl:disjointWith: this expresses that two classes (which are sets) are mutually disjoint. An example:

:Birds rdfs :subClassOf owl :Class.
:Mammals rdfs:subClassOf owl:Class.
:Birds owl:disjointWith :Mammals

{{:c1 a owl:Class. :c2 a owl:Class. :i1 a :c1. :i2 a :c2. :c1 owl:disjointWith :c2.} log:implies {:i1 owl:differentIndividualFrom :i2}} a log:Truth; log:forAll :c1, :c2, :i1, :i2.

<u>Resolution based inference engines</u>

Introduction:

The resolution method was invented in 1965 by J.Allen Robinson. In 1972 Prolog was developed by Alain Comerauer. Prolog uses a subset of FOL (First Order Logic) but resolution engines for full first order logic

exist (e.g. Otter). Resolution is a method for proof finding in logic. Given a set of facts and axioms the resolution mechanism finds a proof of a lemma. The mechanism is complete for FOL i.e. given a lemma the resolution method will deduce its validity or invalidity.

Logical principles

[LOGPRINC]

A theory is *decidable* iff there is an algorithm which can determine whether or not any sentence r is a member of the theory. If a theory is *undecidable* it is not in general possible to decide whether a sentence r is valid or not. *Semi-decidable* means that if a proof can be found eventually it will be found (but after how much time?) but if a proof can not be found there might be no answer (the algorithm can loop).

Gödels completeness theorem:

If *T* is a set of axioms in a first-order language, and a statement *p* holds for any structure *M* satisfying *T*, then *p* can be formally deduced from *T* in some appropriately defined fashion. This amounts to saying that FOL is semi-decidable.

The absence of *contradiction* (i.e., the ability to prove that a statement and its negative are both true) in an axiomatic system is known as *consistency*.

Gödels incompleteness theorem:

Gödel's incompleteness theorem states that all *consistent* axiomatic formulations of *number theory* include undecidable propositions. Another formulation: any formal system that is interesting enough to formulate its own consistency can prove its own consistency *iff* it is inconsistent.

Validity: a set of statements is valid if, for any possible model, it does not contain a contradiction.

Completeness: a logic system is complete if, when a statement is true, it can be proven to be true.

Soundness: a logic system is sound whenever a statement is proven, this statement is also true (semantically).

Decidability, validity, completeness and soundness are all notions reposing on the semantic interpretation of the system e.g. if an algorithm is not sound, it must be proven that certain results of it are not true which can only be stated semantically.

An *interpretation* of FOL (informally) maps a constant to an element of a domain, a predicate to the values {true, false} and a function to an element of the domain. An example is given for the interpretation of formulas:
I(a and b) = I(a) and I(b).

A *model* of a set of formulas is an interpretation that makes every wwf in the set true.

If an interpretation makes each wff (well formed formula) of a set of wwf have the value true, then we say that this interpretation *satisfies* the set of wffs. A wff T *logically follows* from a set of wffs A, if every interpretation satisfying A also satisfies T.

A logic system has the property of *monotonicity* if the addition of new wwfs does not change the truth value of previous derivations.

<u>Mechanism</u>

This is in its simplest form the resolution rule:
```
  A v B
~A v C
=======
  B v C
```

In order to use this mechanism FOL statements are reduced to clause normal form with the following algorithm.

Clause form – algorithm[CENG]:

1) Eliminate the implication signs (A ➔ B becomes (not A) v B)
2) Reduce the scope of the negation signs ((not A ^ B) becomes (not A v not B) etc...)
3) Standardize the variables: rename variables so that each quantifier has its own variable.
4) Eliminate the existential quantifiers by replacing the variables they control by constants or *skolem* functions (ForAll y forSome x P(x,y) becomes forAll y p(g(y), y)) .
5) Convert to *prenex form* by placing all the universal quantifiers at the beginning (which can be done by virtue of the renaming in

point 3). The list of quantifiers at the beginning is called the *prefix*; the rest of the formula is called the *matrix*.

6) Put the matrix in *conjunctive normal form* i.e. as a series of conjunctions (use A v (B ^ C) = (A v B) ^ (A v C)
7) Eliminate the universal quantifiers; all variables are now bound universally.
8) Eliminate the conjunctions obtaining a set of disjunctions (*clauses*).

In *skolemnization* all existential variables are replaced by a function of the universal variable in whose scope they are. Thus in the example higher *x* is replaced by *g(y)*. Such replacement is always possible, as in a model, for each value of y from the domain D will correspond a value of x.

In notation 3 exists a conjunction of triples. Take following n3 database:
:a :b :c.
:f :g :h.
{{:a :x :c. :f :g :h} log:implies {:d :y :z}} log:forAll :x, :y; log:forSome :z.
In prolog this could be:
Triple(a, b, c).
Triple(f, g, h).
Triple(d, Y, Z) :- Triple(a,X,c), Triple(f, g, h).

This is automatically in clause normal form. But what about: log:forSome :z? Must this existential variable not be skolemized?
Let's translate this to first order logic:
forAll X: Triple(a,X,c) and Triple(f,g,h) ➔ forAll Y forSome Z Triple(d,Y,Z). What does Triple(d,Y,Z) really mean? It emans that there is an element of the domain d that whenever it has a property Y there is at least one element of the domain that is a value for that property. What does the inference engine do? It searches values that satisfy Triple(d,Y,Z). If one value is found for Z ok, if two values are found, ok, etc.. In fact the log:forSome can just be ignored and treated as log:forAll because anyhow the search is for values that satisfy the relations and not necessary all values. In fact the log:forAll is treated as a log:forSome if not all solutions are given.

This transformation into Clause Normal Form is always possible for first order logic.

The Herbrand Universe is a general interpretation of a set of clauses from which any other interpretation can be derived. If H(W) is the Herbrand Universe of the set of clauses W then:

- all the constant letters appearing in W are in H(W). If there are no constant letters in W, one allows an arbitrary constant letter α to be in H(W).
- If t1, t2, ...,tn in H(W), then f(i,n)(t1,t2,...,tn) in H(W) with f(i,n) a function letter appearing in W.
- No other elements are in W

Example: database:
Jim, brother_of(X), Wise(X), Taller(X,Y).
The Herbrand universe is:
Jim, brother_of(Jim), brother_of(brother_of(Jim)), ...
As the example shows the Herbrand Universe is infinite if a functor with arity greater than 0 exists.

What is the Herbrand Universe of a triple database? This is composed of:

- all the constant letters: all possible URI's.

Well, thats it!! There are no functions in N3; only triples Triple(...) which is the equivalent of a predicate.

The Herbrand Base includes all possible formulas evaluated on the Herbrand Universe.
Definitions: a literal is an atomic formula or its negation.
A ground instance is obtained by substituting all the variables in a literal by expressions not involving variables.
The Herbrand base of a set of clauses W is the set of all ground instances of all atomic formulas appearing in W, where H(W) is used to obtain expressions not involving variables.

The Herbrand base of the above example is:
Wise(Jim), Wise(brother_of(Jim)), Taller(Jim, Jim),
Taller(brother_of(Jim), Jim) etc...

What is the Herbrand Base of an N3 database?
Example:
:a :b :c.
:f :g :h.
{{:a :x :c. :f :g :h} log:implies {:d :y :z}} log:forAll :x, :y; log:forSome :z.

The Herbrand Base will be:
{:a :b :c. :f :g :h. {:a :uriX1 :c. :f :g :h} ➔ {:d :uriX2 :uriX3}} where uriXn is a uri from the domain and if the log:forSome is neglected.

But is this simple view still valid when the restrictions imposed by rdfs en owl are imposed? These restrictions will have as a consequence that certain uri's cannot be used for constituting the Herbrand Base so that the Herbrand Base will be smaller.

[GENESERETH] A Herbrand interpretation has three parts:

1) the domain is the Herbrand Universe
2) the constants are mapped onto themselves
3) a mapping R from the Herbrand base to {true, false}.

An interpretation of the example above:
Jim => Jim
brother_of(Jim) => brother_of(Jim)
Wise(Jim) => true
Wise(brother_of(Jim)) => false
Taller(Jim, Jim) => false
Taller(brother_of(Jim), Jim) => true
etc ...

An interpretation of the N3 example above:
:a :b :c. => :a :b :c.
:f :g :h. => :f :g :h.
{:a :b :c. :f :g :h} ➔ {:d :uri1 :uri2} => true
{:a :b :c. :f :g :h} ➔ {:d :uri3 :uri4} => false
{:a :uri5 :c. :f :g :h} ➔ {:d :uri6 :uri7} => false

Herbrand theorem: a formula in clause normal form is unsatisfiable iff all of its Herbrand interpretations are false. Furthermore it is unsatisfiable iff some finite conjunction of Herbrand ground instances is unsatisfiable. Hence the Herbrand method: add negation of conclusion to the premises to form the satisfaction set. Loop over Herbrand interpretations. Cross out each interpretation that does not satisfy the sentences in the satisfaction set. If all Herbrand interpretations are crossed out,by the Herbrand Theorem, the set is unsatisfiable.
This procedure is sound and complete if there are only finitely many Herbrand interpretations.

If the domain (of uri's) in N3 is finite then the number of Herbrand interpretations is finite too. If the use of numbers is permitted (and it is)

then the domain is infinite. However many applications will de facto use a finite domain.

Unification and substitution

A substitution subst is a set of ordered pairs:
{(t1,u1),(t2,u2),...} such that (i /= j) ➔ u(i) /= u(j); t(i) are terms and u(i) are variables. The variables are substituted by the terms.
A unifier subst of a set of literals {L(i)} is a most general unifier (mgu) if for any other unifier substx, there exists a substitution substy such that:
L(i)subst.substx =
L(i)substy for all i.
The unification algorithm for finding the mgu for two literals:
Scan the literals till a disagreement is found; the disagreement set consists of the two disagreed symbols; the substitution is enlarged to accomodate the disagreement set. This can be done, if in the disagreement set there exists a variable which can be set to a term. Otherwise failure is reported.

The resolution rule

Given two clauses L and M with no variables in common. Be *l* a term in L and *m* a term in M. Suppose that a mgu subst exists which unifies the set *l* union *not m.* Then the two clauses resolve to a new clause (L – l) union (M-m). The newly inferred clause is called the resolvent.

The resolution method

Given a set of axioms A and a theorem T.

- put the set of axioms A in its conjunctive normal form.
- put *not T* in its conjunctive normal form.
- Form the set of clauses A union *not T.*
- Apply the resolution rule.

If the empty clause is produced, then the theorem T logically follows from the set of axioms A.

Resolution is *refutation complete* for first order logic: if a contradiction exists it will be found.

Resolution strategies

[UMBC]

- Breadth first
- Set of support: at least one parent clause must be from the negation of the goal or one of the “descendents” of such a goal 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. This is not generally complete, but complete for Horn clauses.
- 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 KB''.
- Linear resolution: this is an extension of input resolution. Use P and Q if P is in the initial KB and query or P is an ancestor of Q. This is complete.
- 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.
- Subsumption: eliminate all clauses that are subsumed. This simplifies but does not change the final result.

Paramodulation: is an inference rule for resolution whereby equals are replaced by equals:

$$\frac{\begin{array}{l} t=s \vee K1 \vee \ldots \vee Kn \\ L(t') \vee N1 \vee \ldots \vee Nm \qquad \text{and } \sigma(t') = \sigma(t) \end{array}}{\sigma(L(s)) \vee \sigma(N1) \vee \ldots \vee \sigma(Nm) \vee \sigma(K1) \vee \ldots \vee \sigma(Kn)}$$

<u>Completeness and soundness of the N3Engine</u>

The algorithm used in the engine in this thesis is a resolution algorithm. Solutions are searched by starting with the axiom file and the negation of the lemma(query). When a solution is found a constructive proof of the lemma has also been found in the form of a contradiction that follows from a certain number of traceable unification steps. Thus the soundness of the engine can be concluded.
There are discussions on the internet about whether Notation 3, SWAP and OWL are fol or hol logic see e.g. http://lists.w3.org/Archives/Public/www-rdf-logic/2002Aug/0029.html. So in this thesis the question is asked about the completeness of the engine without giving an answer.
The unification algorithm clearly is decidable as a consequence of the simple ternary structure of N3 triples. The unification algorithm in the module Unify.hs can be seen as a proof of this.

Description of the inference engine N3Engine

1) Structure of the engine

The engine is composed of the following modules:

- N3Parser: this is a parser for Notation 3.
- LoadTree: the load module transforms the parser output into an XML tree. All abbreviations are resolved.
- GenerateDB: merges several input files into one XML data structure. Variables are marked by special tags (see further).
- Unify: this is the module that takes care of the unification. From this module the modules Builtins and Typing are called as extensions of the basic engine.
- N3Engine: this is a lightweight inference engine. It contains a backtracking resolution engine.
- Builtins: this module contains some builtins like owl:DifferentIndividualFrom and owl:List.
- Typing: this is a test whereby each atom recieves a type and only atoms of the same type can be unified.
- Xml: the module that deals with the XML tree.
- Utils: a module that contains some utilitarian constructions.

Fig. ... gives a graphical overview of the modules.

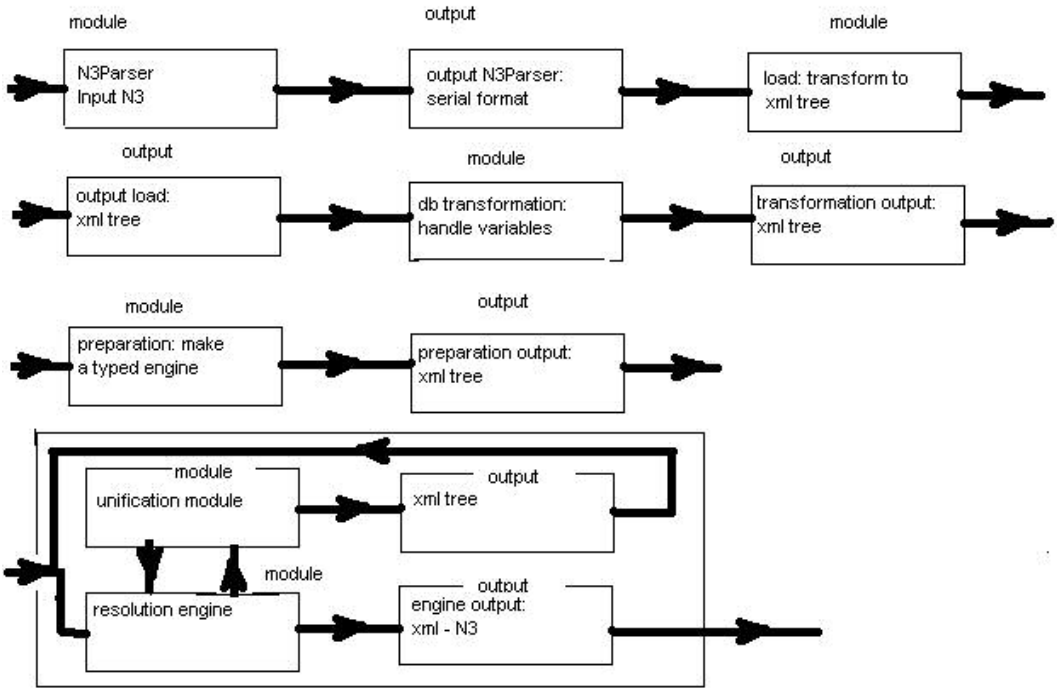


Fig. ... : the module structure of the engine.
The modules Builtin and Typing are called from the unification module.

Development principles

Writing complex programming systems is no easy task and a methodical way of working is well indicated. Here are the principles that were followed while designing the engine:

The program is split in as much modules as possible or sensible. The output of each module is made as simple and verifiable as possible. For the parser it is a simple streaming format, for the load modules, the database module and the transformation module (see further) the output is an xml-tree. For the final engine the output is xml or N3. All outputs are defined with a Backus-Naur form (to be found in the source code).

Each module consists of a set of functions. The functions are kept as small as possible. Each function is tested immediately when it is made, in Haskell with an extra test function, in Python sometimes with a single instruction , mostly also with a test function. When all functions are made the whole module is subjected to the a battery of test cases including those of the Agfa-site [DEROO].

2) The N3Parser

The parser is based on the grammar included in the annexes at the back and has been token from: http://2001/blindfold/sample/n3.bnf and the parser uses also the structures defined in N3 primer: http://www.w3.org/200/10/swap/Primer.htm.
The output data structure consists basically of triples (id,short value, full value) e.g.
Verb/@/:w3cmember/@/<authen#w3cmember>/@/. The value of the verb is a URI; in the N3 source the URI is abbreviated as :w3cmember. The parser (using the prefixes) gives also the complete URI. The tree structure of N3 is kept intact, but dummy subjects and verbs are introduced where these are missing (_subject and _verb).

How does the parser work?
The N3-file is read into a string (by the function readN3) . The parser then starts to “attack” this string. Leading spaces are always skipped. The mechanism is always the same: a function recieves the input string, a

token is read, separation characters are thrown away, then the token is returned, the output string wich is the input string without the token and the thrown away characters and also a boolean flag is returned. If the flag is false then instead of the token an error message is returned and the input string is returned unchanged. After that the input string is synchronized on the next point in the string. Often a lookahead is done to see whether a certain closing character is present or not.
In the module Utils are some general functions for parsing:
(Note: some parameters are not indicated as they are not relevant for the discussion)
checkCharacter(c, s): checks whether a character c is present or not in the input string s. Returns a boolean value.
takec(c,s): will take a given character c from the input string s and returns the rest string and the flag True or False.
parseUntil(a, b): will take from string b until char a is met and returns (True ,c, b-c) where c is the part of b before a with a not included or it returns (False, "", b).
skipTillCharacter(c,s): skip the input string s till the character c is met.The rest of the string is returned without the character.
skipBlancs(s):skips the blancs in a string s and returns the stripped string.
parseComment(s): parses the comment from input s: a line starting with #. Returns the input string without the comment.
startsWith(s1, s2): looks if the input string s1 starts with a certain chain s2. Returns a booelan value.
containsString(s1, s2): test whether the first string s1 is contained in the second string s2. Returns a boolean value.
parseString(a,b): will parse string a from string b and returns (True, b-a) or (False,b).

In pseudo-language what the parser does looks like this:

```
parse tripleSet;.
while triple in tripleSet do {
  parse subject;
    while property in propertyList do {
       parse verb;
       while object in objectList do {
           parse object;
       }
    }
}
```

The parsing of a subject:

if subject = URI
  parseNode;
else parse tripleSet.

For an anonymous triple there is no subject to parse; but an anonymous subject _T$$$n is inserted. An anonymous subject is treated as an existential variable. Indeed the meaning of e.g. [a agg:company]. is that there exists a certain entity that is an agg:company.

Now some comment follows on the most important functions of the parser:
Note: those funcions have an input string and an output string as *input* because they can be called recursively; so both input and outptu have to be passes to the recursively called function.

parseN3(sin, sout): top level of the parser. It takes the inputstring sin (without leading spaces) and returns a string sout consisting of a list of identifier-value pairs separated by the separator e.g. Subject/@/:a/@/Verb/@/:b/@/Object/@/:c if the separator is /@/. This function will put the prefixes (@prefix = description of the namespaces) in a list.
parseTripleSet(sin, sout): parse a set of triples : insert "Set " in the outputstream; parse a subject (parseSubject) and call parsePropertyList. Then call recusively parseTripelSet ; then insert "EndOfSet ". Returns the output string sout and the rest of the input string sin.
parseTriple(sin, sout): parses a singel triple.
parseAnonSet(sin, sout): parse a set of anonymous triples : insert "AnonSet " and call parsePropertyList. Then call recusively parseAnon ; then insert "EndOfSet ". As there is no subject, it is not parsed either.
parsePropertyList(sin, sout) : parses a verb and then calls parseNodeList .
parseProperty(sin, sout): parses a single property.
parseNodeList(sin, sout): parses nodes separated by “,” .Subject and verb are retrieved from a global list.
parseSubject(sin, sout): parse a subject.
parseVerb(sin, sout): parse a verb.
parseObject(sin, sout): parse an object.
The last three functions basically just call the following function.
parseNode(sin) : function for parsing nodes.Input is the string to be parsed; it returns a multiple that exists of the node name, the value of the node and the rest string .
 Formats of a node (=URI):
 <#...> : a local reference.

<> : this page.
:... : a reference in this document
prefix:... : an abbreviation of a namespace.
<URI> : a complete URL.
"..." : a constant
This parses the most basic entities. There are also functions for each of the different nodetypes which are called by parseNode.

Here are some details about what is done to the N3 input (see also the N3 primer referenced above):
- Points are eliminated.
- In ":a is :b of :c" of is eliminated and the verb :b is preceded by "Reverse" meaning subject and object have to be reversed.
- In ":a has :b of :c" has and of are eliminated.
- Anonymous nodes get an anonymous subject value = _T$$$1 ... _T$$$n where n is the index of the last anonymous node e.g. [a agg:company]. becomes _T$$$1 a agg:company.
- The parser is basically recursive descent with look-ahead features.
- When an error occurs the stream is synchronized on the next point and an error message is included in the stream.
- With thanks to Mark P.Jones for his inspiring prolog interpreter and the ideas about parsing included.

I give here a bnf of the output of the parser:

```
ParserOutput ::= Triple (ParserOutput)*|
                 TripleSet (ParserOutput)*|
                 AnonSet (ParserOutput)*

Triple ::= "Triple" Sep Subject Verb Object

AnonTriple ::= AnonSubject Verb Object

TripleSet ::=  Sep Triple* "EndOfSet" Sep

AnonSet ::= Sep AnonTriple* "EndOfSet" Sep

Subject ::= "Subject" Sep String Sep|
            "Set" Sep TripleSet|
            "AnonSet" Sep AnonSet

AnonSubject ::= "Subject" Sep "_T$$$" n Sep
```

Verb ::= ["Reverse" sep] "Verb" Sep String Sep|
"Set" Sep TripleList|
"AnonSet" Sep AnonSet
Note: Reverse means subject and object must be reversed.

Object ::= "Object" Sep String Sep|
"ObjectSet" Sep TripleSet|
"AnonSet" Sep AnonSet

n ::= (digit)*

Sep ::= Separator

Prefix ::= "Prefix" Sep String Sep

The separator is defined in the source code; might be : /@/
_subject and _verb refer to the latest subject and verb.

Now follows an example from an input to the parser and the output.The example is from Jos De Roo. It is clear that the output is not very readable but it it is not meant to be read by humans.

Input:

```
# $Id: authen.axiom.n3,v 1.2 2001/10/01 00:12:34
amdus Exp $

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

<mailto:jos.deroo.jd@belgium.agfa.com> :member
<http://www.agfa.com>.
<http://www.agfa.com> :w3cmember
<http://www.w3.org>.
<http://www.agfa.com> :subscribed <mailto:w3c-ac-
forum@w3.org/>.

{{:person :member :institution.
  :institution :w3cmember <http://www.w3.org>.
  :institution :subscribed :mailinglist}
log:implies
{:person :authenticated :mailinglist}} a
log:Truth; log:forAll :person, :mailinglist,
:institution.
```

Output:

Prefix/@/@prefix log: <http://www.w3.org/2000/10/swap/log#>./@/Prefix/@/@prefix : <authen#>./@/Subject/@/<mailto:jos.deroo.jd@belgium.agfa.com>/@/mailto:jos.deroo.jd@belgium.agfa.com/@/Verb/@/:member/@/<authen#member>/@/Object/@/<http://www.agfa.com>/@/http://www.agfa.com/@/Subject/@/<http://www.agfa.com>/@/http://www.agfa.com/@/Verb/@/:w3cmember/@/<authen#w3cmember>/@/Object/@/<http://www.w3.org>/@/http://www.w3.org/@/Subject/@/<http://www.agfa.com>/@/http://www.agfa.com/@/Verb/@/:subscribed/@/<authen#subscribed>/@/Object/@/<mailto:w3c-ac-forum@w3.org/>/@/mailto:w3c-ac-forum@w3.org//@/Set/@/Set/@/Subject/@/:person/@/<authen#person>/@/Verb/@/:member/@/<authen#member>/@/Object/@/:institution/@/<authen#institution>/@/Subject/@/:institution/@/<authen#institution>/@/Verb/@/:w3cmember/@/<authen#w3cmember>/@/Object/@/<http://www.w3.org>/@/http://www.w3.org/@/Subject/@/:institution/@/<authen#institution>/@/Verb/@/:subscribed/@/<authen#subscribed>/@/Object/@/:mailinglist/@/<authen#mailinglist>/@/EndOfSet/@/Verb/@/log:implies/@/http://www.w3.org/2000/10/swap/log#implies/@/Set/@/Subject/@/:person/@/<authen#person>/@/Verb/@/:authenticated/@/<authen#authenticated>/@/Object/@/:mailinglist/@/<authen#mailinglist>/@/EndOfSet/@/EndOfSet/@/Verb/@/a/@/http://www.w3.org/1999/02/22-rdf-syntax-ns#type/@/Object/@/log:Truth/@/http://www.w3.org/2000/10/swap/log#Truth/@/_subject/@/Verb/@/log:forAll/@/http://www.w3.org/2000/10/swap/log#forAll/@/Object/@/:person/@/<authen#person>/@/_subject/@/_verb/@/Object/@/:mailinglist/@/<authen#mailinglist>/@/_subject/@/_verb/@/Object/@/:institution/@/<authen#institution>/@/

3) The load module

This module transforms the output of the parser into a XML tree. The transformation is straightforward. Here follows the BNF of the output of the load module:

```
LoadOutput ::=   “<DB>”
                 TripleSet*
                 “</DB>”|
                 “<Prefixes>”
                 prefix*
```

“</Prefixes>”
TripleSet ::= “<TripleSet>” Triple* “</TripleSet>”
Embedded_TripleSet ::= Triple*
Triple ::= “<Subject>”
URI|Embedded_TripleSet
“<Verb>”
URI|Embedded_TripleSet
“<Object>“
URI|Embedded_TripleSet
“</Object></Verb></Subject>”
URI ::= “<URI>” UriDesignation “</URI>”
UriDesignation ::= String

One can see here that the complex structure of Notation 3 and the somewhat less complex structure of the parser output are reduced here to a fairly simple structure. Nevertheless this simple structure permits a great expressivity in making declarative statements.

The reason for the tag <TripleSet> will be explained when the example danb.n3 will be explained.
The mechanisms of this module are similar to the parser in the sense that the different constituents of a triple are called recursively.

For the presentation of a triple in XML a hierarchical structure has been chosen. This might seem strange at first sight but this is caused by the abbreviations of N3 , property list and object list. Following presentation was also possible:

```
<subject> ... </subject>
   <propertylist>
      <property>
         <objectlist>
            <object> ... </object>
         </objectlist>
      </property>
         <property> ... </property>
   </propertylist>
```

but here also the structure is partially hierarchical and there are more tags i.e. a more complex structure and less easy for the programmer. Anyhow it is meant to be a structure for use by a computer; the output destined for humans should be in Notation 3 or an even more convivial language. Of course if al the abbreviations are taken away an output without hierarchy is possible.

```
<triple>
  <subject> ... </subject>
  <property> ... </property>
  <object> ... </object>
</triple>
```

All abreviations are resolved so that after this step only sets of complete triples remain. (Recall that anonymous triples did not exist anymore and were already taken away by the parser.) For rules this extension is not done as in that case it is not very interesting. Rules have a special fixed format.
!! not implemented yet !! Optionally by a flag embedded triples can be instantiated as stand-alone triples i.e. suppose there is a subject composed of a set of triples then all of these embedded triples will be instantiated a independent triples. If the user then makes a query that unifies with one of these triples he will recieve extra information (but of course he has to be aware that the search is not done in the original database but in a semantically different database.).

Overview of the functions of the load module:

saveEngine(filename): parse a file with name filename; transform the parsed file to an xml tree in a specific format and save the result to a file.
loadDB(s): transform the parsed file in string format (s) into an XML tree.
addPrefixToTag(tree): reads the prefixes from the prefix tree *tree* and adds them to the XML-tree.(The prefixes are saved in a separate tree during execution and the added to the general tree at the end).
loadString(s, prefixList, xmlTree): prefixes are added to the prefixlist and triples to the xmlTree. Depending on the input s loadString will call different functions:

- the token = “Prefix”: the prefix is added to the prefix tree immediately.
- the token = “Subject”: the function loadTriple is called.
- the token is “Set”: the function loadTerm is called.
- The token is “AnonSet”: the function loadTerm is called.

loadTerm(s, xmlTree): depending on the input s loadTerm will call different functions:

- the token = “Set”: loadTriple is called and then loadTerm is called recursively for handling the next triple or the end of the set.A tag Set is added to the XML-tree.

- the token = “AnonSet”: the same as for “Set”. A tag AnonSet is added to the XML-tree.
- the token is “Subject”: the function loadTriple is called.
- The token is “EndOfSet”: the function loadTriple is called. A tag EndOfSet is added to the XML-tree.

loadTriple(s): this function calls loadSubject and then calls loadPropertyList.
loadPropertyList(s, xmlTree): calls loadProperty; if the following token is “_subject” then loadProperty is called again, else the function returns.
loadProperty(s): calls loadVerb and then calls loadObjectList.
loadObjectList(s, xmlTree): call loadObject; if the next two tokens are “_subject” and “_verb”
then loadObjectList is called again, else the function returns.
loadSubject(s): depending on the input s loadSubject will call different functions:

- the token = “Subject”: a tag Subject with its content is added to the XML-tree.
- the token = “Set”: the Set is loaded and added to the subject.
- the token is “AnonSet”: the same as for Set.

loadVerb(s): depending on the input s loadVerb will call different functions:

- the token = “Verb”: a tag Verb with its content is added to the XML-tree.
- the token = “_subject”: the “_subject” is skipped and a tag Verb is added to the XML-tree.
- the token = “Set”: the Set is loaded and added to the verb.
- the token is “AnonSet”: the same as for Set.

loadObject(s): depending on the input s loadObject will call different functions:

- the token = “Object”: a tag Object with its content is added to the XML-tree.
- the token = “_subject”and the enxt token = “_verb”: the “_subject” and the “_verb” is skipped and a tag Object is added to the XML-tree.
- the token = “Set”: the Set is loaded and added to the object.
- the token is “AnonSet”: the same as for Set.

extendSubjects: extend propertylists (e.g. :a :b :c; :d :e; :f :g.) into separate triples (example becomes: :a :b :c. :a :d :e. :a :f :g.).
Input is an xml tree; output is a list of trees.
extendVerb: expand objectlists (e.g. :a :b :c, :d, :e.) into separate triples (example becomes: :a :b :c. :a :b :d. :a :b :e.).

Here is the output of the load module generated by the previous example:

```
<?xml version="1.0"?>
<DB>
  <TripleSet>
  <Subject>
    <URI>
      <mailto:jos.deroo.jd@belgium.agfa.com>
mailto:jos.deroo.jd@belgium.agfa.com
    </URI>
    <Verb>
      <URI>
        :member <authen#member>
      </URI>
      <Object>
        <URI>
          <http://www.agfa.com>
http://www.agfa.com
        </URI>
      </Object>
    </Verb>
  </Subject>
 </TripleSet>
 <TripleSet>
  <Subject>
    <URI>
      <http://www.agfa.com> http://www.agfa.com
    </URI>
    <Verb>
      <URI>
        :w3cmember <authen#w3cmember>
      </URI>
      <Object>
        <URI>
          <http://www.w3.org> http://www.w3.org
        </URI>
      </Object>
    </Verb>
  </Subject>
 </TripleSet>
 <TripleSet>
  <Subject>
    <URI>
      <http://www.agfa.com> http://www.agfa.com
    </URI>
```

```
    <Verb>
      <URI>
        :subscribed <authen#subscribed>
      </URI>
      <Object>
        <URI>
          <mailto:w3c-ac-forum@w3.org/>
mailto:w3c-ac-forum@w3.org/
        </URI>
      </Object>
    </Verb>
  </Subject>
 </TripleSet>
 <TripleSet>
  <Subject>
    <Subject>
      <Subject>
        <URI>
          :person <authen#person>
        </URI>
        <Verb>
          <URI>
            :member <authen#member>
          </URI>
          <Object>
            <URI>
              :institution <authen#institution>
            </URI>
          </Object>
        </Verb>
      </Subject>
      <Subject>
        <URI>
          :institution <authen#institution>
        </URI>
        <Verb>
          <URI>
            :w3cmember <authen#w3cmember>
          </URI>
          <Object>
            <URI>
              <http://www.w3.org>
http://www.w3.org
            </URI>
          </Object>
        </Verb>
```

```
      </Subject>
      <Subject>
        <URI>
          :institution <authen#institution>
        </URI>
        <Verb>
          <URI>
            :subscribed <authen#subscribed>
          </URI>
          <Object>
            <URI>
              :mailinglist <authen#mailinglist>
            </URI>
          </Object>
        </Verb>
      </Subject>
      <Verb>
        <URI>
          log:implies
http://www.w3.org/2000/10/swap/log#implies
        </URI>
        <Object>
          <Subject>
            <URI>
              :person <authen#person>
            </URI>
            <Verb>
              <URI>
                :authenticated
<authen#authenticated>
              </URI>
              <Object>
                <URI>
                  :mailinglist
<authen#mailinglist>
                </URI>
              </Object>
            </Verb>
          </Subject>
        </Object>
      </Verb>
    </Subject>
    <Verb>
      <URI>
        a http://www.w3.org/1999/02/22-rdf-
syntax-ns#type
```

```
      </URI>
      <Object>
        <URI>
          log:Truth
http://www.w3.org/2000/10/swap/log#Truth
        </URI>
      </Object>
    </Verb>
    <Verb>
      <URI>
        log:forAll
http://www.w3.org/2000/10/swap/log#forAll
      </URI>
      <Object>
        <URI>
          :person <authen#person>
        </URI>
      </Object>
      <Object>
        <URI>
          :mailinglist <authen#mailinglist>
        </URI>
      </Object>
      <Object>
        <URI>
          :institution <authen#institution>
        </URI>
      </Object>
    </Verb>
  </Subject>
 </TripleSet>
</DB>
```

This is quit long but has a simple structure. This is correct XML but this no RDF anymore.

The GenerateDB module

This module takes care of *the transformation* of the input from the load module into the database used in the engine.
The atoms which have a tag “URI” and are variables have their tag changed to “Var” for a universal variable and to “EVar” for an existential variable if they are local variables; if they are global variables the tags will be respectively “GVar” for universal and “GEVar” for existential

variables. Anonymous subjects of the form _T$$$n have their “URI”-tag also changed to “Evar” as they are really existential variables, however only in the query. In the axiom files they mean: there is some subject with this property so it is permitted to suppose the existence of an atom _T$$$n. This is existential operator elimination. All local variables are prefixed with a number which is unique within a block = a set of triples.

Input is the output from the module Load; eventually several load structures are fused.
The output has three parts per input file:
the prefix list, the list of variables and the triple database.

BNF for rules (in N3):

```
 rule ::= "{" triplelist verbimplies triplelist "}" ["a" objectTruth ";"]
          verbforall|verbforsome objectforall
ruleSubject ::= triplelist
triplelist ::= "{" triple* "}"
triple ::= as usual
verbimplies ::= "<http://www.w3.org/2000/10/swap/log#implies>"
objectTruth ::= "<http://www.w3.org/2000//10/swap/log#Truth>"
verbforall ::= "<http://www.w3.org/2000/10/swap/log#forall>"
verbforsome ::= "<http://www.w3.org/2000/10/swap/log#forsome>"
objectforall ::= URI ["," URI]*
```

BNF of the database:

```
database ::= clause*
clause ::= rule | tripleset
tripleset ::= triple*
triple ::= subject verb object [number] [ref1] [ref2]
subject ::= triplelist | "<subject>" content "</subject<"
content ::= URI | var | vare | gvar |gevar
** The first string is the abreviated URI; the second is the full URI.
** For a var the tag uri is simply changed into the tag var.
** var is an universal local variable; vare is an existential local variable
** gvar is a universal global variable; gevar is an existential global
variable.
URI ::= "<URI>" String String "</URI>"
var ::= "<Var>" String String "<Var>"
vare ::= "<EVar>" String String "<EVar>"
gvar ::= "<GVar>" String String "<GVar>"
gevar ::= "<GEVar>" String String "<GEVar>"
verb ::= "<Verb>" content "</Verb>"
object ::= triplelist | "<Object>" content "</Object>"
```

```
triplelist ::= triple*
rule ::= "<Rule> <Subject>" triplelist  "<Verb> <URI>"
        log:implies <http://www.w3.org/2000/10/swap/log#implies>
        "</URI> <Object>" triple "</Object> </Verb> </Subject>"
        ["<Verb> <URI>"
         a http://www.w3.org/1999/02/22-rdf-syntax-ns#type
        "</URI> <Object> <URI>"
         log:Truth http://www.w3.org/2000/10/swap/log#Truth
        "</URI> </Object> </Verb>]
         (<Verb> <URI>"
         log:forAll http://www.w3.org/2000/10/swap/log#forAll
        "</URI>" objectlist "</Verb>)?
         (<Verb> <URI>"
         log:forSome http://www.w3.org/2000/10/swap/log#forSome
        "</URI>" objectlist "</Verb>)?
        </Rule>"
objectlist ::= ("<Object>"  content "</Object>")*
```

It is supposed that the prefix log is used for the SWAP space (however the engine does not use the prefix but the complete URI).

On the scope of variables:
If variables are declared with a separate triple like:
this log:forAll :a, :b, :c.
their scope is global. Beware!! Global variables can give unattented results with a resolution engine.
When they are declared within a tripleset their scope is local.
There are existential and universal variables giving following variable tags: Var, EVar, GVar and GEVar.
Anonymous variables (_T$$$X) have type EVar in the query but not in axiom-files (otherwise they could provoke non-grounded atoms).

Functions for the preparation of the database

mergeInput(inputfiles): the input files are merged into one database. Their origin might be from different internet sites. The last file is the query file who can contain several sets of triples. Each set of triples constitutes a single question to the database.

getVariables(tree): this makes a list of all variables in a database (XMLTree).

markAllVariables(tree): here tha tag of the variables which is “URI” is changed to the specific variable tag: “Var”, “Evar”, “Gvar” or “GEVar”. This is for easy processing by the inference engine.

4) The unification module:

Unification is done on the level of triple sets i.e. a collection of triples.

Functions for substitutions

applySubstitution(substitution, xmlTree): apply a substitution to a XML tree.

A substitution is represented as a list of tuples (in Haskell):
type Subst = [(term, term)] where each term can be a variable, a URI or a tripleset. The composition of two substitutions is just the merge of two lists.
applySubstitutionToList (subst, treeList): apply a substitutino to a list of trees.
showSubstitution(substitution): transform a substitution to a printable string.
showSubstitutionList(substitutionList): transform a list of substitutions to a printable string.

Unification

unifyWithRule(tripleSet, rule): unify a block with a rule. The first clause is a tripleSet.
The second is a rule following the bnf for rules given higher. The last returned parameter is the list of newly generated goals.

unifyTwoTripleSets (tripleSet1, tripelSet2):
unify the triples in two triplesets. The mechanism is: each triple in tripelSet1 must match with a triple in tripleSet2.

unifyTripleWithTripleSet(triple, tripleSet): the triple must match with one of he triples in the tripleSet.

unifyTwoTriples(tripel1, triple2): unify two triples; returns a boolean value indicating the succes and if successfull a substitution; if not successfull the null substitutiuon.

unifyAtoms(atom1, atom2, substitution): unify two atoms (subject, verb or object). s*ubstitution* is the currently valid substitution. Returns a substitution and a boolean. Possibilities: URI with URI;(G) (E)VAR with URI; (G)(E)VAR with(G) (E)VAR, tripelSet with tripleSet; tripelSet with variable.

unifyTwoTerms(term1, term2): this function unifies two terms; each term is a list of triples. Returns a boolean value and a substitution.

unifyTripleWithTerm(triple, term): unify a triple with a term. A term is a list of triples. Returns a boolean value and a substitution.

unifyVerbs: unify a verb with a list of verbs.

unifyObjects: unify an object with a list of objects.

5) <u>The resolution engine</u>

The basic ideas for a resolution engine that works with Notation 3 input were developed by De Roo [DEROO] with the Euler program. Notation 3 has the same semantics as RDF and is just another notation. In order to make proof deduction and verification necessary there is need for logical primitives on top of RDF. This basic logic is defined in the SWAP space [SWAP].

A simple example will demonstrate what it is all about. The example is from Jos De Roo. Some extra comments are added to the original.

```
# $Id: authen.axiom.n3,v 1.2 2001/10/01 00:12:34
amdus Exp $

# The prefixes are the definitions of the
namespaces.
@prefix log:
<http://www.w3.org/2000/10/swap/log#>.
@prefix : <authen#>.


# The following rule has the meaning: if a person
is a member
# of an institution and if that institution is a
member of
# the W3C and it is subscribed to the mailinglist
then
```

```
# this person is authenticated to acces the
mailinglist.
{{:person :member :institution.
  :institution :w3cmember <http://www.w3.org>.
  :institution :subscribed :mailinglist}
log:implies
{:person :authenticated :mailinglist}} a
log:Truth; log:forAll :person, :mailinglist,
:institution.

# From the rule above and the facts hereunder can
be deduced
# that Jos De Roo is authenticated for the
mailinglist from the W3C.
<mailto:jos.deroo.jd@belgium.agfa.com> :member
<http://www.agfa.com>.
<http://www.agfa.com> :w3cmember
<http://www.w3.org>.
<http://www.agfa.com> :subscribed <mailto:w3c-ac-
forum@w3.org/>.
```

Above is what is called the axiom file: composed of facts and rules. This axiom file constitutes the database when read into the engine. A query can then be made by using the following query file:

```
# $Id: authen.lemma.n3,v 1.3 2001/10/15 22:40:11
amdus Exp $

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

# Here the engine is asked to give all persons
that are
# authenticated for the W3C mailinglist.
_:who :authenticated <mailto:w3c-ac-
forum@w3.org/>.
```

Now the engine will resolve this by unifying triples. First the following two triples will be unified clearly giving the substitution {(-:who, :person), (<mailto:w3c-ac-forum@w3.org/>, :mailinglist)}.

```
:person :authenticated :mailinglist.
_:who :authenticated <mailto:w3c-ac-
forum@w3.org/>.
```

As the unification was with the consequent of a rule then the antecedents of the rule will be added to the list of goals. Those antecedents are the following triples:

```
:person :member :institution.
:institution :w3cmember <http://www.w3.org>.
:institution :subscribed :mailinglist.
```

Now the engine will unify the first of these triples with the “data” triple:
`<mailto:jos.deroo.jd@belgium.agfa.com> :member <http://www.agfa.com>.`
giving the substitution {(:person, <mailto:jos.deroo.jd@belgium.agfa.com>), (:institution, <http://www.agfa.com>)}.
Remember that :person, :institution and :mailinglist are variables.
Next `:institution :w3cmember <http://www.w3.org>.`
will be unified with
`<http://www.agfa.com> :w3cmember <http://www.w3.org>.`
giving the substitution {(:institution, :w3cmember)}.
Finally `:institution :subscribed :mailinglist.`
Will be unified with
`<http://www.agfa.com> :subscribed <mailto:w3c-ac-forum@w3.org/>.`
Giving the substitution {(:institution, <http://www.agfa.com>),(:mailinglist, <mailto:w3c-ac-forum@w3.org/>)}.

At this moment following substitutions were made:

{(-:who, :person), (<mailto:w3c-ac-forum@w3.org/>, :mailinglist)}.

```
{(:person,
<mailto:jos.deroo.jd@belgium.agfa.com>),
(:institution, <http://www.agfa.com>)}.
{(:institution, :w3cmember)}.
{(:institution,
<http://www.agfa.com>),(:mailinglist,
<mailto:w3c-ac-forum@w3.org/>)}.
```

When applied to the query this is what happens to the variable : _ :who.

-:who ➔ :person ➔ <mailto:jos.deroo.jd@belgium.agfa.com> so the answer to the query is: <mailto:jos.deroo.jd@belgium.agfa.com>.

This unification process seems to be a simpler than the unification in e.g. Prolog. In the above example simple variables and URI's are matched with each other. However there are some complications.
In the first place a subject or a verb or an object in a triple can be composed i.e. it can be a tripleset instead of an atom (URI or variable). Thus if {:a :b _:x} must be unified with {:a :b { :c :d :e.}} the variable _:x will be replaced by the tripleset {:c :d :e}. In practice this will not happen often but it is possible.
A further complication is illustrated by the following example provided by Dan Brickley, here a little abbreviated. The example is to be found on the Agfa-site [AGFA].The axiom-file is:

```
# $Id: danb.n3,v 1.2 2001/10/01 00:12:35 amdus
Exp $

@prefix agg:
<http://example.com/xmlns/aggregation-demo#> .
@prefix web: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> .

 [      a agg:Company;
             agg:corporateHomepage
<http://megacorp.example.com/>;
             agg:name "MegaCorp Inc.";
             agg:owner  [
                 a agg:Person;
                 agg:name "Mr Mega";
                 agg:personalMailbox
<mailto:mega@megacorp.example.com>;
                 agg:personalHomepage
<http://megacorp.example.com/~mega>;
                 agg:age "50" ];
             agg:ticker "MEGA" ].

 [      a agg:Company;
             agg:corporateHomepage
<http://gigacorp.example.com/>;
             agg:name "GigaCorp Inc.";
             agg:owner  [
                 a agg:Person;
                 agg:name "Mr Giga";
                 agg:personalMailbox
<mailto:giga@gigacorp.example.com>;
```

```
                  agg:personalHomepage
<http://gigacorp.example.com/~mega>;
                  agg:age "46" ];
             agg:ticker "GIGA" ].
```

and the query-file is:

```
# $Id: danb-query.n3,v 1.2 2001/10/01 00:12:35
amdus Exp $

# http://rdfweb.org/2001/01/design/smush.html
# (Q1) What are the technology interests of
persons who own companies that have an ethical
#     policy committment to the policy stated in
the document
#
http://dotherightthing.example.org/policy.xhtml

@prefix agg:
<http://example.com/xmlns/aggregation-demo#>.
@prefix : <danb#>.

this log:forSome :hp, :mb.

[ a agg:Company; agg:corporateHomepage :hp;
agg:owner [ a agg:Person; agg:personalMailbox
:mb]].
```

Here follows by way of example a listing of the first block in the axiom-file without the abbreviations:

```
_T$$$1 a agg:Company.
_T$$$1 agg:corporateHomepage
<http://megacorp.example.com/>.
_T$$$1 agg:name "MegaCorp Inc.".
_T$$$1 agg:owner {_T$$$2 a agg:Person.
                  _T$$$2 agg:name “Mr Mega”.
                  _T$$$2 agg:personalMailbox
<mailto:mega@megacorp.example.com>.
                  _T$$$2 agg:personalHomepage
<http://megacorp.example.com/~mega>.
                  _T$$$2 agg:age “50”. }
_T$$$1 agg:ticker “MEGA”.
```

Instantiations of the anonymous objects are added in the form _T$$$n.

Here is a complex structure where the “[]” stand for anonymous triples i.e. triples with an anonymous subject. The “;” serves further for repeating this anonymous subject in different triples. In the axiom-file there are thus three blocks between “[“ and “]” each containing different triples. To complicate even more some triples have an object that is composed by a block of anonymous triples. In one block the “,” is used to make a set of triples that have subject and verb in common but have each a different object. There are two blocks in the example; each block describes a company; the corporate home page; the name of the company and the owner of the company. These informations in a block belong together.
Now how is a unification done with this example?
In N3Engine all the above abbreviations are resolved so that only triples and sets of triples rest. If unification is now done on the level of a triple then following answer is possible:
[a agg:Company; agg:corporateHomePage <http://megacorp.example.com/> ; agg:owner [a agg:Person; agg:personalMailbox <mailto:giga@gigacorp.example.com>]].
where the name of the corporate homepage is from a different company than the mail-address. So the unification has to be done on the level of a block of triples. It probably should be best if such things could be laid down in some specification. This is the reason why the tag <TripleSet> was introduced in the load module so that the N3Engine is able to recognize a block.
Another difficulty with the unification mechanism is caused by looping. Following example will illustrate this.

```
# File ontology1.axiom.n3
# After a suggestion of Jos De Roo

@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#>.

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

rdfs:subClassOf a owl:TransitiveProperty.
:mammalia rdfs:subClassOf :vertebrae.
:rodentia rdfs:subClassOf :mammalia.
:mouse rdfs:subClassOf :rodentia.
:piep rdfs:subClassOf :mouse.
```

```
# File ontology1.query.n3
# After a suggestion by Jos De Roo
@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.
```

The solution is (by N3Engine.050802.hs):

```
:mammalia rdfs:subClassOf :vertebrae.
:rodentia rdfs:subClassOf :vertebrae.
:mouse rdfs:subClassOf :vertebrae.
:piep rdfs:subClassOf :vertebrae.
```

Now what does this do? In the axiom-file rdfs:subClassOf is declared to be a owl:TransitiveProperty. In the facts a subclass hierarchy is declared; all subclasses mentioned are subclasses of :vertebrae. The engine finds the right solution. So what is the problem?
The problem lies in the declaration of what a transitive property is:

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

In this rule are the triples:

```
:c1 :p :c2.
:c2 :p :c3.
:c1 :p :c3.
```

Now those triples unify with every other triple as they only contain variables. This causes a combinatorial explosion in the engine. The goals created by the rule: :c1 :p :c2. :c2 :p :c3. will match with all other triples in the database causing numerous new paths in the dept-first search of the resolution engine. The solution to this might be: making the transitivity of subClassOf explicit or working with a typed engine: see further the text on typed resolution engine.

<u>The implementation of ontological restraints by use of a typed resolution engine</u>

In RDFS a basic ontology is introduced as an extension to RDF. Further work on ontology is done by the WebOnt working group of the W3C

[WEBONT]. Such an ontology imposes a classification as well as restrictions on RDF-data. In the following a general scheme for implementing such ontologies in a resolution engine as well as a specific scheme for the inference engine N3Engine based on N3 are discussed.
A resolution engine generally consists of a database of clauses on the one hand and a query on the other hand where solutions are found by resolution. The resolution is done by the unification of terms and the sustitution of variables.
The implementation of an ontology can be done by attaching types to the atoms of the database e.g. in a Prolog-like way:
SubClassOf(Vertebrae, Mammalia).
SubClassOf(Mammalia, Rodentia).
TransitiveProperty(SubClassOf).
Because we define SubClassOf to be a transitive property the query SubClassOf(Vertebrae,Rodentia) should be positive.But how do we define TransitiveProperty in Prolog?
In N3 this is defined by:
{:p a :TransitiveProperty. :a :p :b. :b :p :c.} log:implies { :a :p :c.}; log:forAll :a, :b, :c, :p.
This cannot be done in Prolog as quantification over a property is not possible. The engine N3Engine can work with such a definition. If the following axiom is given:

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

and the following query is done:

```
?:who rdfs:subClassOf :vertebrae
```

the answer will be:

:mammalia rdfs:subClassOf :vertebrae.

```
:rodentia rdfs:subClassOf :vertebrae.
```

Note: “a” is translated to rdf:type.
Note: relevant namespaces:

```
The site where the experimental logics for the
semantic web are defined :
@prefix log:
<http://www.w3.org/2000/10/swap/log#> .
```

```
The site for the ontology defined by the WebOnt
working group :
@prefix owl: <http://www.w3.org/2002/07/owl#> .
The XML Schema definitions:
@prefix xsd: <http://www.w3.org/2001/XMLSchema#>
.
The rdf Schema definitions:
@prefix rdfs: <http://www.w3.org/2000/01/rdf-
schema#> .
The rdf syntax definition:
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-
syntax-ns#> .
```

However this method is not practical: the triples :c1 :p :c2. :c2 :p :c3. :c1 :p :c3. can be unified with all other triples as :c1, :c2, :c3 and :p are all variables. This then will provoke a combinatorial explosion. A possible solution is to let the engine produce a rule:
{:a rdfs:subClassOf :b. :b rdfs:subClassOf :c.} log:implies {:a rdfs:subClassOf :c}; log:forAll :a, :b, :c.
Here the triples will only be unified with other triples who have the predicate rdfs:subClassOf thereby countering the combinatorial explosion. The rule for transitive property must be dropped from the database.

Another solution is typing. All atoms in the database are given a type and implicitly all terms possess a (composed) type. When unification takes place only terms of the same type can be unified. How is this done in the previous example?
All atoms belong automatically to a superclass e.g. called :resource. Thus :vertebrae will be of the type :resource. :mammalia will have the type :vertebrae. :rodentia will have the type :mammalia. However clearly it must be possible to unify :rodentia with a variable of type :vertebrae. So the unification engine must take into account the class hierarchies. So the need is for: typing rules and rules for typed unification.
This gives the following model for implementation in N3Engine taking into account the triple structure (subject - predicate(or verb) – object) where subject, predicate or verb are either URI's or variables. Variables represent URI's. Subject, predicate and object are commonly named atoms. To implement the typing structure every atom is tagged with type information that permits to define the type and the unification of types. The tagging is to be taken literally as the data-structure inside N3Engine is an XML-structure. The typing and unification rules are described in a N3-file. The engine determines the types and matching following the rules layed down in this N3-file. The execution of those rules is done by

formulating a query that is executed by the engine itself. Suppose there are two atoms t1 and t2 each with their type information , let's say type1 and type2. So the engine might issue a query: :type1 :unification :type2 against the axiom-file typing_rules.n3. If the answer to that query is positive then the two types match, if not the two types do not match.
As an example the following rules might be in the file typing_rules.n3:

```
# This rule defines a transitive property
# The variable p will recieve the type
# owl:TransitiveProperty; the others will recieve
the type :resource
{{:p a owl:TransitiveProperty. :a :p :b. :b :p
:c.} log:implies {:a :p :c}} a log:Truth;
log:forAll :a, :b, :c, :p.
# rule for atoms with type resource
# probably best to built into the engine???
# :a and :b recieve type resource
{{:a a :resource. :b a :resource.} log:implies
{:a :unification :b}} a log:Truth; log:forAll :a,
:b.
# rule for subClassOf
# all predicates subClassOf will recieve the type
# owl:TransitiveProperty
rdfs:subClassOf :type owl:TransitiveProperty.


# user class definitions
:mammalia rdfs:subClassOf :vertebrae.
:rodentia rdfs:subClassOf :mammalia.
```

The last two rules are added (perhaps temporarily) from the user-input. Then the following query might be issued:

```
:rodentia rdfs:subClassOf :vertebrae.

If the query is: ?:x rdfs:subClassOf :vertebrae.
then ?:x will have the type class. Thus ?:x will
only unify with URI's of type class.
```

This will provoke a unification with :a :p :c. of the owl:TransitiveProperty rule. As :p is a owl:TransitiveProperty the query :rdfs:subClassOf :type owl:TransitiveProperty will be launched and (of course) be answered positively. This query will only be launched once as the type owl:TransitiveProperty will be added to the possible type of rdfs:subClassOf. It follows that the engine must dispose of a list of

atoms with their types and restrictions. The resolution database is built with pointers to the list of atoms. This enhances the efficiency of the engine as now no longer enormous masses of alphanumerical data to have to be manipulated in stacks. Other queries e.g. :bird :has :feathers will not be matched with this rule because :has does not have the owl:transitiveProperty. In this way the combinatorial explosion is stopped.

Other ontological restrictions can be handled in the same way. rdfs:property and rdfs:subPropertyOf can be treated in the same way.

```
# statements about rdfs:propertyOf
# this shows that an atom can have more than one
type.
rdfs:subPropertyOf :type owl:transitiveProperty.
rdfs:subPropertyOf :type rdfs:property .
# rule for subPropertyOf
{{:a rdfs:subPropertyOf :b. :b rdfs:propertyOf
:c.} log:implies {:a rdfs:PropertyOf :c}} a
log:Truth; log:forAll :a, :b,:c.

# user statements
:bird_color rdfs:property :bird.
:wing_color rdfs:subPropertyOf :bird_color.
```

Query: :wing_color rdfs:property :bird.
Here types are not necessary for the unification.

Of course subProperty and property will only unify with subProperty or property.

rdfs:range and rdfs:domain impose restrictions on a property in the sense that things having the property must be of the class indicated by rdfs:range and the values of the property must be of the class rdfs:domain.

```
:wingSize a rdfs:property.
:wingSize rdfs:range :length.
```
:wingSize rdfs:domain :bird.

When the query:
:aquila :wingSize :1.
Is launched the engine will find in its atom-table that wingSize is a rdfs:property with restrictions domain = bird and range = length.

It therefore will launch the queries:
`:aquila a :bird.`
:1 a :length.
and probably get a positive result (but not if the user did not define :aquila to be a bird.)

There are two ways properties and restraints are put in the atom-list:

1) during a preparatory fase the user-input is scanned and types and restraints are determined following the rules in the file preparatory_types.n3.
2) if, during execution, the type of an atom is determined or a restraint is inherited (as a consequence of type determination) these are added to a temporary atom-list that contains the atoms for the blocks in the goal list; this temporary atom-list must be saved on the stack for backtracking purposes. The typing and restriction info for atoms in the database does not change anymore after the preparatory phase.
3) The same principle is valid for variables.

Here are some more examples taken from owl:
[see http://www.agfa.com/w3c/euler/owl-rules]

owl:inverseOf is defined as owl:inverseOf a rdf:property. It defines the 'inversability' of two predicates. This gives the following rule:

{{:p a owl:inverseOf. :q a owl:inverseOf. :p owl:inverseOf :q. :s :p :o.} log:implies {:o :q :s.}} a log:Truth; log:forAll :p, :q,:s, :o.

Here :p and :q will recieve the type owl:inverseOf so :o :q :s. will only match with properties that have the type owl:inverseOf.
This mechanism works also for following owl-items:
owl:samePropertyAs
owl:sameClassAs
owl:equivalentTo

An additional remark is necessary concerning owl:equivalentTo. Indeed instead of working with types for handling equivalences it seems better to eliminate them in a preparatory phase; if it is known that two atoms are equivalent one of the two can be supresses and replaced with the other and the equivalence statement can be deleted. This can also be done if equivalences are concluded during resolution execution.

Conclusion:

1) The use of typing is an important technique in the prevention of combinatorial explosions, certainly when general clauses are used that unify with all other clauses (like the general rules produced by WebOnt).
2) The use of typing also is a means to control correct usage; if a property is used with a certain domain and range the types of subject and object will be controlled and, if not good, no unification will take place.

So typing will enforce the correct usage of properties like class, subClassOf, subPropertyOf etc...

Structure of the engine

The inference engine : this is where the resolution is executed. There are three parts to this engine:

a) *solve* : the generation of new goals by selection of a goal from the goallist by some selection procedure and unifying this goal against all blocks of the database thereby producing a set of *alternative blocks*. If the goallist is empty a solution has been found and a backtrack is done in search of other solutions.
b) *choose*: add one of the alternative blocks to the goallist; the other ones are pushed on the stack. Each set of alternative goals is pushed on the stack together with the current goallist and the current substitution. If solve did not generate any alternative goals there is a failure (unification did not succeed) and a backtrack must be done to get an alternative goal.
c) *backtrack*: an alternative goal is retrieved from the stack and added to the goallist. If the stack is empty the resolution process is finished. A failure occurs if for none of the alternatives a unification is possible; otherwise a set of solutions is given.

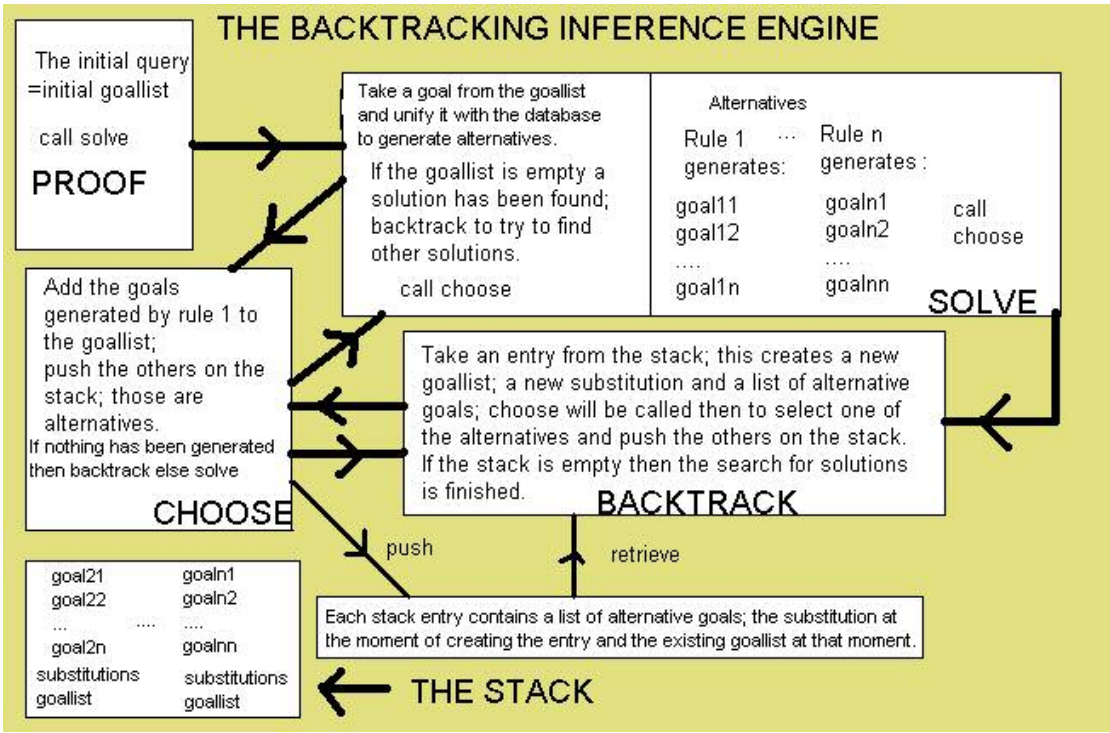


Fig. A schematic overview of the backtracking engine.

What is called above a substitution is a set of transformations of variable and atoms into variables and atoms. For each solution to a query there exists such a set of transformations that will transform the variables in the query into grounded atoms. Thus in fact the answer to the query is a list of substitutions = a list of lists of transformations.

<u>The backtracking resolution mechanism in pseudo-language</u>

Note: a block = a set of triples is what corresponds in prolog to a clause.
goalList = all blocks in the query.
do {
  while (! goalList = empty) {
    select a goal.
    If this goal is a new goal unify this goal against the database producing a set of alternative goals (= all the blocks which unify with the selected goal) and eliminate this goal from the goalList
    else the engine is looping; backtrack to the proper choicepoint.
    add one of this alternative set to the goalList and push the others on the stack
  } // while
  retrieve an alternative from the stack
} *until* (stack == empty)

The goal which is selected from the goalList is the head of the goalList. The alternative which is chosen from the list of alternatives is the first alternative in the list.

The mechanism which is followed is in fact SLD-resolution: Selection, Linear, Definite. There is a selection function for blocks; a quit simple one given the fact that the first in the list is selected (here is one of the point where optimisation is possible namely by using another selection function); linear means that the resolution rule is followed. In prolog the definition of *a definite sentence* is a sentence that has exactly one positive literal in each clause and the unification is done with this literal. In the N3-database rules have exactly one “positive block” which is the consequent and facts always are a positive block.

Following resolution strategies are respected by an SLD-engine:

- *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 goal or one of the “descendents” of such a goal 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. This is not generally complete, but complete for Horn clauses. Is this complete for N3?
- *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 KB's.Is this complete for N3 KB’s?.
- *linear resolution*: the general resolution rule is followed.
- *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.

The data structure of N3Engine:

The module works internally with an XML-tree.

Here follows an overview of the important functions of N3Engine:

Main function

searchProof(filename1, filename2): read a parsed axiom-file filename1, read a parsed lemma-file filename2, execute the refutation algorithm and display the result.
e.g. searchProof "authen.axiom.n3" "authen.lemma.n3"

The resolution engine

proof(database, query): a proof has as input the database of clauses and a query. The output is a list of substitutions (one for each found solution) and an XMLTree that contains information about the resolution process if the verbose flag is set.
The query is the initial goallist.

solve(trace, substitution, goallist, stack, integer, database): search a solution for the goal at the top of the goallist. All triples who unify with this goal are added to the alternative list. For a rule the antecedents of a matched rule are added to the goal list. When the goal list is empty and the stack is empty all solutions have been found; if the goallist is empty but not the stack then a backtrack will be done in search of further solutions.
If a goal fails then retrieve the last saved situation from the stack.
The iput integer counts the number of steps of the engine in view of limiting the maximal number of steps.
Output is a substitution and the trace XMLtree.

choose(trace, substitution, goallist, alternatives_list, stack, integer, database): choose an alternative. In this implementation add the first alternative to the goal list and save the others on the stack together with the last found substitution and the previous goal list. Then call again the solve function. If the list of alternatives is empty then backtrack to the last list. The integer has the same function as with *solve.*Returns a substitution and the trace XMLtree.

backtrack(trace, stack, integer, database): retrieve an alternative from the stack and call the function *choose.* If there is no alternative return an empty substitution and the trace XMLtree. The input integer is as higher.

getAlts(database, clause, substitution): get the list of matches of a tripleset with the heads in the database. This is the kernel of the resolution engine. The first parameter is the database; the second is the goal to unify; the third parameter is the current substitution.

Output is a list of alternatives consisting of pairs that contain a clause and a substitution: type Alt = (XMLTree, Subst), and an XMLTree that contains trace data.

Bibliography

[AIT] Hassan Aït-Kaci *WAM A tutorial reconstruction* .

[BUNDY] Alan Bundy , *Artificial Mathematicians, May 23, 1996,* *http://www.dai.ed.ac.uk/homes/bundy/tmp/new-scientist.ps.gz*

[CASTELLO] R.Castello e.a. *Theorem provers survey*. University of Texas at Dallas. *http://citeseer.nj.nec.com/409959.html*

[CENG] André Schoorl, *Ceng 420 Artificial intelligence* University of Victoria, *http://www.engr.uvic.ca/~aschoorl/ceng420/*

[CHAMPIN] Pierre Antoine Champin, 2001 – 04 –05, *http://www710.univ-lyon1.fr/~champin/rdf-tutorial/node12.html*

[COQ] Huet e.a. *RT-0204-The Coq Proof Assistant : A Tutorial,* *http://www.inria.fr/rrt/rt-0204.html.*

[DAML+OIL] *DAML+OIL (March 2001) Reference Description*. Dan Connolly e.a. W3C Note 18 December 2001. Latest version is available at http://www.w3.org/TR/daml+oil-reference.

[DENNIS] Louise Dennis Midlands Graduate School in TCS, *http://www.cs.nott.ac.uk/~lad/MR/lcf-handout.pdf*

[DESIGN] http://www.w3.org/DesignIssues/
** Tim Berners-Lee's site with his design-issues articles* .

[DEROO] http://www.agfa.com/w3c/jdroo
** the site of the Euler program*

[DICK] A.J.J.Dick, *Automated equational reasoning and the knuth-bendix algorithm: an informal introduction,* Rutherford Appleton Laboratory Chilton, Didcot OXON OX11 OQ, *http://www.site.uottawa.ca/~luigi/csi5109/church-rosser.doc/*

[DONALD] *http://dream.dai.ed.ac.uk/papers/donald/subsectionstar4_7.html*

[GANDALF] http://www.cs.chalmers.se/~tammet/gandalf
** Gandalf Home Page*

[GENESERETH] Michael Genesereth, *Course Computational logic, Computer Science Department, Stanford University.*

[GHEZZI] Ghezzi e.a. *Fundamentals of Software Engineering,* Prentice-Hall 1991 .

[GUPTA] Amit Gupta & Ashutosh Agte, *Untyped lambda calculus, alpha-, beta- and eta- reductions, April 28/May 1 2000,* *http://www/cis/ksu.edu/~stefan/Teaching/CIS705/Reports/GuptaAgte-2.pdf*

[HARRISON] J.Harrison, *Introduction to functional programming, 1997.*

[HILOG] http://www.cs.sunysb.edu/~warren/xsbbook/node45.html

[JEURING] J.Jeuring and D.Swierstra, *Grammars and parsing,* *http://www.cs.uu.nl/docs/vakken/gont/*

[KERBER] Manfred Kerber, *Mechanised Reasoning,* Midlands Graduate School in Theoretical Computer Science, The University of Birmingham, November/December 1999, http://www.cs.bham.ac.uk/~mmk/courses/MGS/index.html

[LAMBDA] http://www.cse.psu.edu/~dale/lProlog/
* *lambda prolog home page*

[LINDHOLM] Lindholm, *Exercise Assignment Theorem prover for propositional modal logics, http://www.cs.hut.fi/~ctl/promod.ps*

[LOGPRINC] http://www.earlham.edu/~peters/courses/logsys/glossary.htm#m]

[MCGUINESS] Deborah McGuiness, *Explaining reasoning in description logics,* 1966 Ph.D.Thesis

[MYERS] CS611 LECTURE 14 *The Curry-Howard Isomorphism,* Andrew Myers.

[NADA] Arthur Ehrencrona, Royal Institute of Technology Stockholm, Sweden, *http://cgi.student.nada.kth.se/cgi-bin/d95-aeh/get/umeng*

[OTTER] http://www.mcs.anl.gov/AR/otter/ Otter Home Page

[OWL Features]
Feature Synopsis for OWL Lite and OWL. Deborah L. McGuinness and Frank van Harmelen. W3C Working Draft 29 July 2002. Latest version is available at http://www.w3.org/TR/owl-features/.

[OWL Issues]
Web Ontology Issue Status. Michael K. Smith, ed. 10 Jul 2002.

[OWL Reference]
OWL Web Ontology Language 1.0 Reference. Mike Dean, Dan Connolly, Frank van Harmelen, James Hendler, Ian Horrocks, Deborah L. McGuinness, Peter F. Patel-Schneider, and Lynn Andrea Stein. W3C Working Draft 29 July 2002. Latest version is available at http://www.w3.org/TR/owl-ref/.

[OWL Rules] *http://www.agfa.com/w3c/euler/owl-rules*

[PFENNING_1999] Pfenning e.a., *Twelf a Meta-Logical framework for deductive Systems,* Department of Computer Science, Carnegie Mellon University, 1999.

[PFENNING_LF] *http://www-2.cs.cmu.edu/afs/cs/user/fp/www/lfs.html*

[RDFM] *RDF Model Theory* .Editor: Patrick Hayes <http://www.w3.org/TR/rdf-mt/>

[RDFMS] *Resource Description Framework (RDF) Model and Syntax Specification* http://www.w3.org/TR/1999/REC-rdf-syntax-19990222

[RDF Primer] *http://www.w3.org/TR/rdf-primer/*]

[RDFSC] *Resource Description Framework (RDF) Schema Specification 1.0* <http://www.w3.org/TR/2000/CR-rdf-schema-20000327>

[SCHNEIER] Schneier Bruce, *Applied Cryptography,*

[STANFORD] Stanford Encyclopedia of philosophy - *Automated reasoning http://plato.stanford.edu/entries/reasoning-automated/*

[SWAP/CWM] http://www.w3.org/2000/10/swap
http://infomesh.net/2001/cwm
CWM is another inference engine for the web .

[TBL] Tim-berners Lee, *Weaving the web,*
[TBL01] Berners-Lee e.a.*The semantic web,* Scientific American May 2001
[TWELF] http://www.cs.cmu.edu/~twelf,Twelf Home Page
[UNCERT]Hin-Kwong Ng e.a. *Modelling uncertainties in argumentation,* Department of Systems Engineering & Engineering Management
The Chinese University of Hong Kong
http://www.se.cuhk.edu.hk/~hkng/papers/uai98/uai98.html
[UMBC] TimothyW.Finin, Computer Science and Electrical Engineering, University of Maryland Baltimore Country,
http://www.cs.umbc.edu/471/lectures/9/sld040.htm
[USHOLD] Michael Ushold The boeing company, *Where is the semantics of the web?*
http://cis.otago.ac.nz/OASWorkschop/Papers/WhereIsTheSemantics.pdf
[VAN BENTHEM] Van Benthem e.a., *Logica voor informatici,*
Addison Wesley 1991.
[WALSH] Toby Walsh, *A divergence critic for inductive proof,* 1/96, Journal of Artificial Intelligence Research, http://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
Important for a thorough introduction in Gofer . In Dutch .
[WOS] Wos e.a. *Automated reasoning,*Prentice Hall, 1984
[W3SCHOOLS http://www.w3schools.com/w3c/w3c_intro.asp]

Abbreviations

ALF	: Algebraic Logic Functional Programming Language
CA	: Certification Authority
CWM	: Closed World Machine An experimental inference engine for the semantic web
DTD	: Document Type Definition , a language for defining XML-objects .
HTML	: Hypertext Markup Language
N3	: Notation 3
OWL	: Ontology Web Language
PKI	: Public Key Infrastructure
RA	: Registration Authority
RDF	: Resource Description Framework
RDFS	: RDF Schema
SweLL	: Semantic web Logic Language
W3C	: World Wide Web Consortium
WAM	: Warren Abstract Machine Probably the first efficient implementation of prolog .
XML	: Extensible Markup Language . The difference with HTML is that tags can be freely defined in XML .

Annexe:

Theorem provers an overview
==========

1.Introduction

As this thesis is about automatic deduction using RDF declarations and rules, it is useful to give an overview of the systems of automated reasoning that exist. In this connection the terms machinal reasoning and, in a more restricted sense, theorem provers, are also relevant.
First there follows an overview of the varying (logical) theoretical basises and the implementation algorithms build upon them. In the second place an overview is given of practical implementations of the algoritms. This is by no means meant to be exhaustive.

Kinds of theorem provers :
[DONALD]
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.

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 serve to generate proofs but to check proofs; those proofs must be in a format that can be easily transported over the net.

2. Overview of principles
2.1. General remarks
Automated reasoning is an important domain of computer science. The number of applications is constantly growing.

* proving of theorems in mathematics
* reasoning of intelligent agents
* natural language understanding
* mechanical verification of programs
* hardware verifications (also chips design)

* planning
* and ... whatever problem that can be logically specified (a lot!!)

Hilbert-style calculi have been trditionally 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]

2.2 Reasoning using resolution techniques: see the chapter on resolution.

2.3. Sequent deduction

[STANFORD] [VAN BENTHEM]
Gentzen analysed the proof-construction process and then devised two deduction calculi for classical logic: the natural deduction calculus (NK) and the sequent calculus (LK).
Sequents are expressions of the form A ➔ B where both A and B are sets of formulas. An interpretation I satisfies the sequent iff either I does not entail a (for soma a in A) or I entails b (for some b in B). It follows then that proof trees in LK are actually refutation proofs like resolution.
A reasoning program, in order to use LK must actually construct a proof tree. The difficulties are:

1) LK does not specify the order in which the rules must be applied in the construction of a proof tree.
2) The premises in the rule $\forall \rightarrow$ and $\exists \rightarrow$ rules inherit the quantificational formula to which the rule is applied, meaning that the rules can be applied repeatedly to the same formula sending the proof search into an endless loop.
3) LK does not indicate which formula must be selected newt in the application of a rule.
4) The quantifier rules provide no indication as to what terms or free variables must be used in their deployment.
5) The application of a quantifier rule can lead into an infinitely long tree branch because the proper term to be used in the instantiation never gets chosen.

Axiom sequents in LK are valid and the conslusion of a rule is valid iff its premises are. This fact allows us to apply the LK rules in either direction, forwards from axioms to conslusion, or backwards from conclusion to axioms. Also with the exception of the cut rule, all the rules'premises are subformulas of their respective conclusions. For the purposes of automated deduction this is a significant fact and we would

want to dispense with the cut rule; fortunately, the cut-free version of LK preserves its refutation completeness(Gentzen 1935). These results provide a strong case for constructing proof trees in backward fashion. Another reason for working backwards is that the truth-functional fragment of cut-free LK is confluent in the sense that the order in which the non-quantifier rules are applied is irrelevant.
Another form of LK is *analytic tableaux.*

2.4. Natural deduction

[STANFORD] [VAN BENTHEM]
In natural deduction (NK) deductions are made from premisses by 'introduction' and 'elimination' rules.
Some of the objections for LK can be applied to NK.

1) NK does not specify in which order the rules must be applied in the construction of a proof.
2) NK does not indicate which formula must be selected next in the application of a rule.
3) The quantifier rules provide no indication as to what terms or free variables must be used in their deployment.
4) The application of a quantifier rule can lead into an infinitely long tree branch because the proper term to be used in the instantiation never gets chosen.

As in LK a backward chaining strategy is better focused.
Fortunately, NK enjoys the subformula property in the sense that each formula entering into a natural deduction proof can be restricted to being a subformula of $\Gamma \cup \Delta \cup \{\alpha\}$, where Δ is the set of auxiliary assumptions made by the not-elimination rule. By exploiting the subformula property a natural dedcution automated theorem prover can drastically reduce its search space and bring the backward application of the elimination rules under control. Further gains can be realized if one is willing to restrict the scope of NK's logic to its intuistionistic fragment where every proof has a normal form in the sense that no formula is obtained by an introduction rule and then is eliminated by an elimination rule.

2.5. The matrix connection method

[STANFORD] Suppose we want to show that from (PvQ)&(P ➔ R)&(Q➔R) follows R. To the set of formulas we add ~R and we try to find a contradiction. First this is put in the conjuntive normal form:

(PvQ)&(~PvR)&(=QvR)&(~R).Then we represent this formula as a matrix as follows:

P	Q
~P	R
~Q	R
~R	

The idea now is to explore all the possible vertical paths running through this matrix. Paths that contain two complementary literals are contradictory and do not have to be pursued anymore. In fact in the above matrix all paths are complementary which proves the lemma. The method can be extended for predicate logic ; variations have been implemented for higher order logic.

2.6 Term rewriting

Equality can be defined as a predicate. In /swap/log the sign = is used for defining equality e.g. :Fred = :human. The sign is an abbreviation for "http://www.daml.org/2001/03/daml+oil/equivalent". Whether it is useful for the semantic web or not, a set of rewrite rules for N3 (or RDF) should be interesting.
Here are some elementary considerations (after [DICK]).
A *rewrite rule,* written E ==> E', is an equation which is used in only one direction. If none of a set of rewrite rules apply to an expression, E, then E is said to be in *normal form* with respect to that set.
A rewrite system is said to be *finitely terminating* or *noetherian* if there are no infinite rewriting sequences E ==> E' ==> E" ==> ...
We define an ordering on the terms of a rewrite rule where we want the right term to be more simpler than the left, indicated by E >> E'. When E >> E' then we should also have: σ(E) >> σ(E') for a substitution σ. Then we can say that the rewrite system is finitely terminating if and only if there is no infinite sequence E >> E' >> E" ... We then speak of a *well-founded ordering.* An ordering on terms is said to be *stable* or *compatible with term structure* if E >> E' implies that
i) σ(E) >> σ(E') for all substitutions σ
ii) f(...E...) >> f(...E..) for all contexts f(...)

If unique termination goes together with finite termination then every expression has just one normal form. If an expression c leads always to the same normal form regardless of the applied rewrire rules and their sequence then the set of rules has the propertry of *confluence*. A rewrite system that is both confluent and noetherian is said to be *canonical*. In essence, the Knuth-Bendix algorithm is a method of adding new rules to a noetherian rewrite system to make ik canonical (and thus confluent).

A *critical expression* is a most complex expression that can be rewritten in two different ways. For an expression to be rewritten in two different ways, there must be two rules that apply to it (or one rule that applies in two different ways).Thus, a critical expression must contain two occurrences of left-hand sides of rewrite rules. Unification is the process of finding the most general common instance of two expressions. The unification of the left-hand sides of two rules would therefore give us a critical expression to which both rules would be applicable. Simple unification, however, is not sufficient to find all critical expressions, because a rule may be applied to any part of an expression, not just the whole of it. For this reason, we must unify the left-hand side of each rule with all possible sub-expressions of left-hand sides. This process is called *superposition*.So when we want to generate a proof we can generate critical expressions till eventually we find the proof. Superposition yields a finite set of most general critical expressions.
The Knuth_Bendix completion algorithm for obtaining a canonical set (confluent and noetherian) of rewrite rules:
(Initially, the axiom set contains the initial axioms, and the rule set is empty)

A **while** the axiom set is not empty **do**
B **begin** Select and remove an axiom from the axiom set;
C Normalise the axiom
D **if** the axiom is not of the form x= x **then**
Begin
E order the axiom using the simplification ordering, >>, to
Form a new rule (stop with failure if not possible);

F Place any rules whose left-hand side is reducible by the new rule
back into the set of axioms;

G Superpose the new rule on the whole set of rules to find the set of
critical pairs;

H Introduce a new axiom for each critical pair;
End
End.

Three possible results: -terminate with success
- terminate with failure: if no ordering is possible in step E
- loop without terminating

A possible strategy for proving theorems is in two parts. Firstly, the given axioms are used to find a canonical set of rewrite rules (if possible). Secondly, new equations are shown to be theorems by reducing both sides to normal form. If the normal forms are the same, the theorem is shown to be a consequence of the given axioms; if different, the theorem is proven false.

2.7. Mathematical induction

[STANFORD]
The implementation of induction in a reasoning system presents very challenging search control problems. The most important of these is the ability to detremine the particular way in which induction will be applied during the proof, that is, finding the appropriate induction schema. Related issues include selecting the proper variable of induction, and recognizing all the possible cases for the base and the inductive steps. Lemma caching, problem statement generalisation, and proof planning are techniques particularly useful in inductive theorem proving.
[WALSH] For proving theorems involving explicit induction ripling is a powerful heuristic developed at Edinburgh. A *difference match* is made between the induction hypothesis and the induction conclusion such that the *skeleton* of the annotated term is equal to the induction hypothesis and the erasure of the annotation gives the induction conclusion. The annotation consists of a *wave-front* which is a box containing a *wave-hole. Rippling* is the process whereby the wave-front moves in a well-defined direction (e.g. to the top of the term) by using directed (annotated) rewrite rules.

2.8. Higher order logic

[STANFORD] Higher order logic differs from first order logic in that quantification over functions and predicates is allowed. In higher order logic unifiable terms do not always possess a most general unifier and higher order unifciation is itself undecidable. Higher order logic is also incomplete; we cannot always proof wheter a given lemma is true or false.

2.9. Non-classical logics

For the different kinds of logic see the overview of logic.
Basically [STANFORD] three approaches to non-classical logic:

1) Try to mechanize the non-classical deductive calculi.

2) Provide a formulation in first-order logic and let a classical theorem prover handle it.
3) Formulate the semantics of the non-classical logic in a first-order framework where resolution or connection-matrix methods would apply.

Automating intuistionistic logic has applications in software development since writing a program that meets a specification corresponds to the problem of proving the specification within an intuistionistic logic.

2.10. Lambda calculus

[GUPTA] The syntaxis of lambda calculus is simple. Be E an expression and I an identifier then E::= (\I.E) | (E1 E2) | I . \I.E is called a lambda abstraction; E1 E2 is an application and I is an identifier. Identifiers are bound or free. In \I.I*I the identifier I is bound. The free identifiers in an expression are denoted by FV(E).
FV(\I.E) = FV(E) – {I}
FV(E1 E2) = FV(E1) U FV(E2)
FV(I) = {I}

An expression E is closed if FV(E) = {}
The free identifiers in an expression can be affected by a substitution.
[E1/I]E2 denotes the substitution of E1 for all *free* occurences of I in E2.
[E/I](\I.E1) = \I.E1
[E/I](\J.E1) = \J.[E/I]E1, if I <> J and J not in FV(E).
[E/I](\J.E1) = \K.[E/I][K/J]E1, if I <> J, J in FV(E) and K is new.
[E/I](E1 E2) = [E/I]E1 [E/I]E2
[E/I]I = E
[E/I]J = J, if J <> I

Rules for manipulating the lambda expressions:

alpha-rule: \I.E ➔ \J.[J/I]E, if J not in FV(E)
beta-rule: (\I.E1)E2 ➔ [E2/I]E1
eta-rule: \I.E I ➔ E, if I not in FV(E).

The alpha-rule is just a renaming of the bound variable; the beta-rule is substitution and the eta-rule implies that lambda expressions represent functions.

Say that an expression, E, containds the subexpression (\I.E1)E2; the subexpression is called a *redex;* the expression [E2/I]E1 is its *contractum;* and the action of replacing the contractum for the redex in

E, giving E', is called a *contraction* or a *reduction step.* A reduction step is written E ➔ E'. A *reduction sequence* is a series of reduction steps that has finite or infinite length. If the sequence has finite length, starting at E1 and ending at En, n>=0, we write E1 ➔ *En. A lambda expression is in *(beta-)normal form* if it contains no (beta-)redexes. Normal form means that there can be no further reduction of the expression.

Properties of the beta-rule:

a) The confluence property (also called the *Church-Rosser* property) : For any lambda expression E, if E ➔ *E1 and E ➔ *E2, then there exists a lambda expression, E3, such that E1 ➔ *E3 and E2 ➔ *E3 (modulo application of the alpha-rule to E3).

b) The uniqueness of normal forms property: if E can be reduced to E' in normal form, then E' is unique (modulo application of the alpha-rule).
There exists a rewriting strategy that always discovers a normal form. The *leftmost-outermost* rewriting strategy reduces the leftmost-outermost redex at each stage until nomore redexes exist.

c) The standardisation property: If an expression has a normal form, it will be found by the leftmost-outermost rewriting strategy.

2.11. Typed lambda-calculus

[HARRISON] There is first a distinction between *primitive types and composed* types. The *function* space type constructor e.g. bool -> bool has an important meaning in functional programming. Type *variables* are the vehicle for polymorphism. The types of expressions are defined by *typing rules*. An example: if s:sigma -> tau and t:sigma than s t: tau. It is possible to leave the calculus of conversions unchanged from the untyped lambda calculusif all conversions have the property of *type preservation*.
There is the important theorem of *strong normalization:* every typable term has a normal form, and every possible sequence starting from a typable term terminates. This looks good, however, the ability to write nonterminating functions is essential for Turing completeness, so we are no longer able to define all computable functions, not even all total ones. In order to regain Turing-completeness we simply add a way of defining arbitrary recursive functions that is well-typed.

2.12. Proof planning

[BUNDY] In proof planning common patterns are captured as computer programs called *tactics.* A tactic guides a small piece of reasoning by specifying which rules of inference to apply. (In HOL and Nuprl a tactic is an ML function that when applied to a goal reduces it to a list of subgoals together with a justification function.) *Tacticals* serve to combine tactics.
If the and-introduction (ai) and the or-introduction (oi) constitute (basic) tactics then a tactical could be:
ai THEN oi OR REPEAT ai.
The *proof plan* is a large, customised tactic. It consists of small tactics, which in turn consist of smaller ones, down to individual rules of inference.
A *critic* consists of the description of common failure patterns (e.g. divergence in an inductive proof) and the patch to be made to the proof plan when this pattern occur. These critics can, for instance, suggest proiving a lemma, decide that a more general theorem should be proved or split the proof into cases.
Here are some examples of tactics from the *Coq* tutorial.
Intro : applied to a ➔ b ; a is added to the list of hypotheses.
Apply H : apply hypothesis H.
Exact H : the goal is amid the hypotheses.
Assumption : the goal is solvable from current assumptions.
Intros : apply repeatedly Intro.
Auto : the prover tries itself to solve.
Trivial : like Auto but only one step.
Left : left or introduction.
Right : right or introduction.
Elim : elimination.
Clear H : clear hypothesis H.
Rewrite : apply an equality.

Tacticals from Coq:
T1;T2 : apply T1 to the current goal and then T2 to the subgoals.
T;[T1|T2|…|Tn] : apply T to the current goal; T1 to subgoal 1; T2 to subgoal 2 etc…

2.13. Genetic algorithms

A “genetic algorithm” based theorem prover could be built as follows: take a axiom, apply at random a rule (e.g. from the Gentzen calculus) and measure the difference with the goal by an evaluation function and so on … The evaluation function can e.g. be based on the number and sequence of logical operators. The semantic web inference engine can easily be adapted to do experiments in this direction.

Overview of theorem provers

[NOGIN] gives the following ordering:

* Higher-order interactive provers:
    - Constructive: ALF, Alfa, Coq, [MetaPRL, NuPRL]
    - Classical: HOL, PVS
* Logical Frameworks: Isabelle, LF, Twelf, [MetaPRL]
* Inductive provers: ACL2, Inka
* Automated:
    - Multi-logic: Gandalf, TPS
    - First-order classical logic: Otter, Setheo, SPASS
    - Equational reasoning: EQP, Maude
* Other: Omega, Mizar

Higher order interactive provers

Constructive

Alf: abbreviation of "Another logical framework". ALF is a structure editor for monommorphic Martin-Löf type theory.

Nuprl: is based on constructive type theory.

Coq:

Classical

HOL: Higher Order Logic: based on LCF approach built in ML. Hol can operate in automatic and interactive mode. HOL uses classical predicate calculus with terms from the typed lambda-calculus = Church's higher-order logic.

PVS: (Prototype Verification System) based on typed higher order logic

Logical frameworks

Pfenning gives this definition of a logical framework [PFENNING_LF]*:*

A *logical framework* is a formal meta-language for deductive systems. The primary tasks supported in logical frameworks to varying degrees are

- specification of deductive systems,
- search for derivations within deductive systems,
- meta-programming of algorithms pertaining to deductive systems,
- proving meta-theorems about deductive systems.

In this sense lambda-prolog and even prolog can be considered to be logical frameworks. Twelf is called a meta-logical framework and could thus be used to develop logical-frameworks. The border between logical and meta-logical does not seem to be very clear : a language like lambda-prolog certainly has meta-logical properties also.

Twelf, Elf: An implementation of LF logical framework; LF uses higher-order abstract syntax and judgement as types. Elf combines LF style logic definition with lambda-prolog style logic programming. Twelf is built on top of LF and Elf.

Twelf supports a variety of tasks: [PFENNING_1999]
Specification of object languages and their semantics, implementation of algorithms manipulating object-language expressions and deductions, and formal development of the meta-theory of an object-language.
For semantic specification LF uses the judgments-as-types representation technique. This means that a derivation is coded as an object whose type represents the judgment it establishes. Checking the correctness of a derivation is thereby reduced to type-checking its representation in the logical framework (which is efficiently decidable).
Meta-Theory. Twelf provides two related means to express the meta-theory of deductive systems: higher-level judgments and the meta-logic M 2 .
A higher-level judgment describes a relation between derivations inherent in a (constructive) meta-theoretic proof. Using the operational semantics for LF signatures sketched above, we can then execute a meta-theoretic proof. While this method is very general and has been used in many of the experiments mentioned below, type-checking a higher-level judgment does not by itself guarantee that it correctly implements a proof.
Alternatively, one can use an experimental automatic meta-theorem proving component based on the meta-logic M 2 for LF. It expects as input a \Pi 2 statement about closed LF objects over a fixed signature and a termination ordering and searches for an inductive proof. If one is found, its representation as a higher-level judgment is generated and can then be executed.

Isabelle: a generic, higher-order, framework for rapid proptotyping of deductive systems [STANFORD].(Theorem proving environments : Isabelle/HOL,Isabelle/ZF,Isabelle/FOL.)

Object logics can be fromulated within Isabelle’s metalogic. Inference rules in Isabelle are represented as generalized Horn clauses and are applied using resolution. However the terms are higher order logic (may contain function variables and lambda-abstractions) so higher-order unification has to be used.

Automated provers

Gandalf : [CASTELLO] Gandalf is a resolution based prover for classical first-order logic and intuistionistic first-order logic. Gandalf implements a number of standard resolution strategies like binary resolution, hyperresolution, set of support strategy, several ordering strategies, paramodulation, and demodulation with automated ordering of equalities. The problem description has to be provided as clauses. Gandalf implements a large number of various search strategies. The deduction machinery of Gandalf is based in Otter's deduction machinery. The difference is the powerful search autonomous mode. In this mode Gandalf first checks whether a clause set has certain properties, then selects a set of different strategies which are likely to be useful for a given problem and then tries all these strategies one after another, allocating less time for highly specialized and incomplete strategies, and allocating more time for general and complete strategies.

Otter : is a resolution-style theorem prover for first order logic with equality [CASTELLO]. Otter provides the inference rules of binary resolution, hyperresolution, UR-resolution and binary paramodulation. These inference rules take a small set of clauses and infer a clause; if the inferred clause is new, interesting, and usefull, then it is stored. Otter maintains four lists of clauses:

. usable: clauses that are available to make inferences.

. sos = set of support (Wos): see further.

. passive

. demodulators: these are equalities that are used as rules to rewrite newly inferred rules.

The main processing loop is as follows:

While (sos is not empty and no refutation has been found)

1) Let given_clause be the lightest clause in sos
2) Move given_clause from sos to usable.
3) Infer and process new clauses using the inference rules in effect; each new clause must have the given clause as one of its parents and members of usable as its other parents; new clauses that pass the retention test are appended to sos.

Other

Prolog

PPTP: Prolog Technology Theorem Prover: full first order logic with soundness and completeness.

Lambda-prolog: higher order constructive logic with types.

Lambda prolgo uses hereditary Harrop formulas; these are also used in Isabelle.

TPS: is a theorem prover for higher-order logic that uses typed lambda-calculus as its logical representation language and is based on a connection type mechanism (see matrix connection method) that incorporates Huet's unification algorithm.TPS can operate in automatic and interactive mode.

LCF (Stanford, Edinburgh, Cambridge) (Logic for Computable Functions):

Stanford:[DENNIS]

- declare a goal that is a formula in Scotts logic
- split the goal into subgoals using subgoaling commands
- subgoals are solved using a simplifier or generating more subgoals
- these commands create data structures representing a formal proof

Edinburgh LCF

- solved the problem of a fixed set of subgoaling by inventing a metalanguage (ML) for scripting proof commands.
- A key idea: have a type called *thm*. The only values of type thm are axioms or are obtained from axioms by applying the inference rules.

Nqthm (Boyer and Moore): (New Quantified TheoreM prover) inductive theorem proving; written in Lisp.

Overview of different logic systems

1. General remarks
2. Propositon logics
3. First order predicate logic
   3.1. Horn logic
4. Higher order logics
5. Modal logic/temporal logic
6. Intuistionistic logic
7. Paraconsistent logic
8. Linear logic

1. General remarks

Logic for the internet does of course have to be sound but not necessarily complete.
Some definitions ([CENG])
First order predicate calculus allows variables to represent function letters.
Second order predicate calculus allows variables to also represent predicate letters.
Propositional calculus does not allow any variables.
A calculus is decidable if it admits an algorithmic representation, that is, if there is an algorithm that, for any given Γ and α, it can determine in a finite amount of time the answer, "Yes" or "No", to the question "Does Γentail α?".
If a wwf (well formed formula) has the value true for all interpretations, it is called *valid*. Gödel's undecidability theorem: there exist wff's such that their being valid can not be decided. In fact first order predicate calculus is semi-decidable: if a wwf is valid, this can be proved; if it is not valid this can not in general be proved.
[MYERS].
The Curry-Howard isomorphism is an isomorphism between the rules of natural deduction and the typing proof rules. An example:

```
A1    A2                          e1: t1    e2:t2
--------------                    --------------------
A1 & A2                             <e1, e2>:t1*t2
```

This means that if a statement P is logically derivable and isomorph with t then there is a program and a value of type t.

<u>Proof and programs</u>

In logics a proof system like the Gentzen calculus starts from assumptions and a lemma. By using the proof system (the axioms and theorems) the lemma becomes proved.
On the other hand a program starts from data or actions and by using a programming system arrives at a result (which might be other data or an action; however actions can be put on the same level as data). The data are the assumptions of the program or proof system and the output is the lemma. As with a logical proof system the lemma or the results are defined before the program starts. (It is supposed to be known what the program does). (In learning systems perhaps there could exist programs where it is not known at all moments what the program is supposed to do).
So every computer program is a proof system.

A logic system has characteristics *completeness* and *decidability*. It is complete when every known lemma can be proved.(Where the lemma has to be known sematically which is a matter of human interpretation.) It is decidable if the proof can be given.
A program that uses e.g. natural deduction calculus for propositional logic can be complete and decidable. Many contemporary programming systems are not decidable. But systems limited to propositional calculus are not very powerful. The search for completeness and decidability must be done but in the mean time, in order to solve problems (to make proofs) flexible and powerful programming systems are needed even if they do not have these two characteristics.

2. Propositon logics
3. First order predicate logic
Points 2 , 3 mentioned for completeness. See [BENTHEM].
3.1. Horn logic: see above the remarks on resolution.

4. Higher order logics

_____
[DENNIS].
In second order logic quantification can be done over functions and predicates. In first order logic this is not possible.
Important is the Gödel *incompleteness theorem :* the corectness of a lemma cannot always be proven.
Another problem is the Russell paradox : P(P) iff ~P(P). The solution of Russell : disallow expressions of het kind P(P) by introducing typing so P(P) will not be well typed.
In the logic of computable functions (LCF) terms are from the typed lambda calculus and formulae are from predicate logic. This gives the LCF family of theorem provers : Stanford LCF, Edinburgh LCF, Cambridge LCF and further : Isabelle, HOL, Nuprl and Coq.

5. Modal logic (temporal logic)

[LINDHOLM] [VAN BENTHEM]Modal logics deal with a number of possible worlds, in each of which statements of logic may be made. In propositional modal logics the statements are restricted to those of propositional logic. Furthermore, a reachability relation between the possible worlds is defined and a modal operator (usually denoted by), which define logic relations between the worlds.
The standard interpretation of the modal operator is that P is true in a world x if P is true in any world that is reachable from x, i.e. in all worlds

y where R(x,y). P is also true if there are no successor worlds. The operator can be read as « necessarily ».
Usually, a dual operator ◊ is also introduced. The definition of ◊ is ~ ~, translating into « possible ».
There is no « single modal logic », but rather a family of modal logics which are defined by axioms, which constrain the reachability relation R. For instance, the modal logic **D** is characterized by the axiom P ➔ ◊P, which is equivalent to reqiring that all worlds must have a successor.
A proof system can work with semantic tableaux for modal logic.
The addition of and ◊ to predicate logic gives *modal predicate logic*.

Temporal logic: in temporal logic the possible worlds are moments in time. The operators and ◊ are replaced by respectively by G (going to) and F (future). Two operators are added : H (has been) and P (past).
Epistemic logic: here the worlds are considered to be knowledge states of a person (or machine). is replaced by K (know) and ◊ by ~K~ (not know that = it might be possible).
Dynamic logic: here the worlds are memory states of a computer. The operators indicate the necessity or the possibility of state changes from one memory state to another. A program is a sequence of worlds. The accessibility relation is T(π) where π indicates the program. The operators are : [π] and <π>. State b entails [π] φ iff for each state accesible from b φ is valid. State b entails <π> if there is a state accessible with φ valid. φ ➔ [π] ψ means : with input condition φ all accessible states will have output condition ψ (correctness of programs).
Different types of modal logic can be combined e.g. [π] K φ ➔K [π]φ which can be interpreted as: if after execution I know that φ implies that I know that φ will be after execution.
S4 or provability logic : here is interpreted as a proof of A.

6. Intuistionistic or constructive logic

[STANFORD] In intuistionistic logic the meaning of a statement resides not inits truth conditions but in the means of proof or verification. In classical logic p v ~p is always true ; in constructive logic p or ~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)

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 transform a hypothetical proof of A into a proof of a contradiction.

7. Paraconsistent logic
[STANFORD]
The development of paraconsistent logic was initiated in order to challenge the logical principle that anything follows from contradictory premises, *ex contradictione quodlibet (ECQ)*. 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 is 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.
Relevant logics.

<u>8.Linear logic</u>
[LINCOLN] Patrick Lincoln Linear Logic SIGACT 1992 SRI and Stanford University.
In linear logic propositions are not viewed as static but more like ressources.
If D implies (A and B) ; D can only be used once so we have to choose A or B.
In linear logic there are two kinds of conjuntions: *multiplicative conjunction* where A ⊗ B stands for the proposition that one has both A and B at the same time; and *additive conjuntion* A & B stands for a choice between A and B but not both. The *multiplicative disjunction* written A ℘ B stands for the proposition “if not A, then B” . Additive disjunction A ⊕ B stands for the possibility of either A or B, but it is not

known which. There is the *linear implication* A –o B which can be interpreted as “can B be derived using A *exactly once”*.
There is a *modal storage operator*. !A can be thought of as a generator of A’s.
There exists propositional linear logic, first order linear logic and higher order linear logic. A sequent calculus has been defined.
Recently (1990) propositional linear logic has been shown to be undecidable.

Varia:

Introducing programming language features in Notation 3:

The case study about the travel agent shows that the inference engine has a complex task to accomplish: determining paths and scheduling itineraries. Though, no doubt, a lot can be accomplished using facts and rule sets, the existence of programming language features could be a great asset.
An hypothetical example: a number n has to be multiplied 5 times by a number n1. This gives us a function with two variables. A function (or procedure) will be called with a query. The result returned will be a substitution. (Also possible should be to return a fact that is added to the database).
The definition of the procedure:

```
{:procedure :definition :multiply_5_times.
 :params = ?p1, ?p2, ?p3.
# ?p3 is the(a) return parameter.
?temp :assign “5”.
:while {?temp math:greater “0”} {
   ?p1 = {?p1 math:multiply ?p2}.
   ?temp = {?temp math:subtract “1”}.
}.
?p3 :substitution ?p1.
} # end of procedure :multiply_5_times
```

The query:

```
{:procedure :query :multiply_5_times.
  : params = “6”, “7”, ?r}.
```

Of course all this has to be interpreted by the inference engine.
The result should be: 6*7*7*7*7*7.

The translation from Haskell to SWeLL

Take the following Haskell function:

```
test (a,b)
  | a > c = b
  | b > c = a
  where c = b - a
```

Each item of the case statement is translated to a separate rule in SWeLL. The declarations after the where keyword are just facts in the SWeLL rule.

```
{this log:forAll :a,:b,:c.
:c math:diff :b, :a..
{:a math:greater :c.} log:implies {:b :test :a, :b}.
{:b math:greater :c} log:implies {:a :test :a, :b}}.
```

and the function is called with:
_:what :test "5", "4". in a query or in a rule:
{:what :test “5”, “4”. :what math:equal “5”.}log:implies {:whatever :is :whatever}; log:forAll :what.

which would give two solutions:
"5" and "4".
The difference with Haskell is that Haskell gives only one solution: the case item are executed sequentially and the first match is returned. It should not be too difficult to instruct the engine to keep only the first solution...

Or in Prolog:

```
MathDiff(b,a,c).
Test(a,b,b) :- MathGreater(a,c).
Test(a,b,a) :- MathGreater(b,c).
```

With query:
Test(a,b,X).

This can be done with functions (or procedures) written in whatever language e.g. the same example in Python:

```
def test(a,b):
        c = b-a
        if a > c:
                return b
        elif b > c:
                return a
```

with of course the same result in Notation 3.

As complete complex programs can be written with functions in this style (see the Haskell modules of this thesis) this can be seen as a general way to write programs in Notation 3 by making a specification in Haskell.

N-ary predicates

A triple like :s :p :v. could be interpreted like a binary predicate: :p(:s, :v) Or like a ternary predicate like Triple(:s, :p, :v). The form :p(:s, :v) is not really completely equivalent to :s :p :v. as the predicate :p acquires a special status and is not anymore on the same level as :s and :v. However following the rdf specification the property or predicate really has a special status. So for the sake of this discussion the form :p(:s, :v) will be used.
:p(:s, :v) then represents a binary predicate. How about a unary predicate (like not). This is simple: [:not :a] is the negation of :a. Now this really is a triple with an anonymous subject. The N3Engine transforms this to something like: _T$$$1 :not :a. The _T$$$1 is treated as an existential variable when in a query and as grounded term when in an axiom-file (the reasons for this are explained elsewhere in this thesis.).
How about a null-ary predicate (or a fact) like e.g. Venus. [:Venus _T$$$1] could do. Is this a needed thing? Probably not.
A ternary predicate could be put this way:
[:p :o1, :o2, :o3]. The N3Engine transforms this to:
_T$$$1 :p :o1.
_T$$$1 :p :o2.
_T$$$1 :p :o3.
This works because the unit of unification in N3Engine is a tripleset and the two triples stay together within the tripleset.Take the unary predicates [:p :o1], [:p :o2] and [:p :o3]. This will reduce to:
_T$$$1 :p :o1.
_T$$$2 :p :o2.
_T$$$3 :p :o3.
thus not giving the same result as [:p :o1, :o2]. Take as an example:
[:house :price, :size, :color] as a ternary predicate and [:house :location] as an unary predicate and the facts:
[:house “10000”, “100”, “yellow”].
[:house “Brussels”].
and the query:
[:house ?a, ?b, ?c].

This query will only match with [:house “10000”, “100”, “yellow”] because unification is done on the level of triplesets.
However the query:
[:house ?a] will give two answers:
[:house “10000”] and [:house “Brussels”] a rather confusing result.
So what? Don’t give the same name to predicates of different arities.

Global, local, existential and universal variables

Reflexion API

On looping problems

Two mechanisms for avoiding looping have been built into the engine:

1) When a goal generates alternatives a check is done whether any of the alternatives after substitution return the original goal. This alternative is discarded.
2) A list of goals is kept. Whenever a solution is found this list of goals is emptied. When a goal is presenting itself for the second time a backtrack is done to the level of the first occurrence of this goal. As the original goal does no longer exist one of the alternatives of the original goal will be chosen as the new goal.

The dictionary

When a goal is matched against the database a search is done against all clauses of the database. Suppose the database contains 2000 clauses but only 20 clauses can really unify with the goal. Then 1980 clauses are being tried to match for nothing. So a dictionary is made of atoms and the clause list where the atom can be found. A list of the atoms of the goal is made and then all clauses that contain these atoms are searched in the dictionary.
Some clauses match with every goal: they should always be included.
A refinement: in the dictionary is made a difference between an occurrence as subject, property or verb.