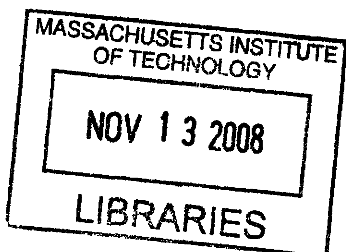
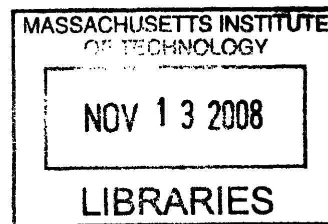
Reasoning Strategies for Semantic Web Rule Languages



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 13 2008
LIBRARIES



MASSACHUSETTS INSTITUTE OF TECHNOLOGY
NOV 13 2008
LIBRARIES

by

Joseph Scharf

S.B. C.S., M.I.T., 2007

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 2008

© Massachusetts Institute of Technology 2008. All rights reserved.

Author ....
Department of Electrical Engineering and Computer Science
August 22, 2008

Certified by ....
Tim Berners-Lee
Professor, Electrical Engineering and Computer Science
Thesis Supervisor

Certified by ....
Lalana Kagal
Research Scientist, Computer Science and Artificial Intelligence Lab
Thesis Reader

Accepted by ....
Arthur C. Smith
Chairman, Department Committee on Graduate Theses

ARCHIVES

Reasoning Strategies for Semantic Web Rule Languages

by

Joseph Scharf

Submitted to the Department of Electrical Engineering and Computer Science
on August 22, 2008, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Dealing with data in open, distributed environments is an increasingly important problem today. The processing of heterogeneous data in formats such as RDF is still being researched. Using rules and rule engines is one technique that is being used. In doing so, the problem of handling heterogeneous rules from multiple sources becomes important. Over the course of this thesis, I wrote several kinds of reasoners including backward, forward, and hybrid reasoners for RDF rule languages. These were used for a variety of problems and data in a wide range of settings for solving real world problems. During my investigations, I learned several interesting problems of RDF. First, simply making the term space big and well namespaced and the language low enough expressivity did not make computation necessarily easier. Next, checking proofs in an RDF environment proved to be hard because the basic features of RDF that make it possible for it to represent heterogeneous data effectively make proofs difficult. Further work is needed to see if some of these problems can be mitigated. Though rules are useful, using rules correctly and efficiently for processing RDF data proved to be difficult.

Thesis Supervisor: Tim Berners-Lee
Title: Professor, Electrical Engineering and Computer Science

Thesis Reader: Lalana Kagal
Title: Research Scientist, Computer Science and Artificial Intelligence Lab

Acknowledgments

I would like to thank Chris Hanson for the design on AIR and for helping me understand how a reasoner for it had to work.

I would like to thank Tim Berners-Lee for both the development of the Semantic Web as well as for supervising and mentoring me over the last four years. I would like to thank both him and Dan Connolly for the development of Cwm and the Notation3 language.

Thanks to Lalana Kagal for trying to supervise me and hold me to a schedule, as well as being a user helping set the requirements for much of the software used.

Thanks to Gerry Susman for talking to me and helping with writing this thesis.

Thanks to IARPA for funding my research.

Thanks to Dan Connolly, Sean B. Palmer, and all others who have contributed to Cwm over the years.

I would like to thank my parents for all of their help and support over the years.

THIS PAGE INTENTIONALLY LEFT BLANK

Contents

1 Introduction 15

2 Background 17

2.1 RDF Data Model 17

2.1.1 Modeling Data as RDF 18

2.2 RDF Syntaxes 18

2.3 RDFS Reasoning 19

2.4 OWL 19

2.5 Rule Languages 21

2.5.1 N3Logic 21

3 Cwm Reasoner 25

3.1 Introduction 25

3.2 Forward Chained Reasoner 25

3.2.1 Technique 27

3.2.2 Builtins 27

3.3 Backward Chained Reasoner 28

3.4 SPARQL query engine 31

3.5 Proof Generation and Checking 33

4 AIR Reasoner 41

4.1 Introduction 41

4.2 RETE AIR Reasoner 45

4.3 TREAT AIR Reasoner 49
4.4 TMS 51
4.5 Proof Generation 51
4.6 Performance 53
4.7 The Choice of Python 54

5 Related Work **57**
5.1 RIF 57
5.2 Pellet 58
5.3 Euler 58
5.4 WhyNot 58
5.5 InferenceWeb 59
5.6 Python-DLP 59
5.7 Pychinko 59

6 Contribution **61**

7 Conclusion and Lessons Learned **63**

A Example Notation3 Files **67**
A.1 TREAT Example 67
A.2 Inconsistent RDFS 68
A.3 Turing Completeness of Notation3 Rules 68

B Avoiding Redundant Rule Firings **73**

C Notation3 Rules are still Computationally Intractable **75**

D An example AIR file **77**

E base-rules.ttl **81**

F Useless Encoding as RDF **91**

G The Semantics of Alt **93**

H The Notation3 Syntax **97**

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

3-1 A cube of anonymous nodes 38

4-1 Rete Structure for a single pattern 47

4-2 TREAT Structure for a single pattern 50

4-3 How a rule fires after a successful match 52

4-4 The things generated for ARL1 and Prox card policies on sample data. 54

THIS PAGE INTENTIONALLY LEFT BLANK

List of Tables

2.1 The "RunAsPython" builtin 24

3.1 Some very symmetric Notation3 37

4.1 The time and things generated for ARL1 and Prox card policies on sample data. 53

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 1

Introduction

The great power of computers lies in their generality. A single computer can be programmed to do anything, and to process any kind of data. This generality extends to programs and data: The more programs that can process given data, the more useful it is. Also, the more data that a program can process, the more useful it is.

Any data format must be modeled to some level. If this model is too general, it means nothing. If it is too specific, one cannot use it to represent the data one wants. Getting the right balance to get this right is an open problem.

As one extreme, we have the byte-stream representation of a data base. This can represent anything. However, there is no semantics whatsoever associated with the format. Everything is in level above it. Therefore, we can transfer the files in this format and store on the disk, as indeed we do, for data integration this is not useful.

A higher level model of data is the relational database. There are an almost unlimited number of ways of mapping a particular dataset to a set of relations. Some of them are little better than the byte stream. Others model the data in a more extensible way.

The Semantic Web is an attempt to allow for just that. By using Uniform Resource Identifiers [8] (or Internationalized Resouce Identifiers) to represent things, it is possible to merge different data source in sane ways. Then, with a representation of a set of triples, it is possible to model arbitrary tuples. Any data can be represented in this way. Modeled well in RDF, the data may be usable for more than just

the original intention. The W3C Semantic Web Best Practices Working Group[42] publishes some guides on how to model data well.

Once the data is in useful formats, the question becomes what can be inferred from the data, and how to verify the inferences. This is building on decades of work in logic and inference.

In order to do this, we need to define logics and data models that scale to the Web. Notation3 is an attempt to supply both. It is an extension of RDF to include the ability to represent many more things.

One of the things Notation3 can represent is rules. Rules, saying that one thing implies another, have been created in many forms over the decades. Several algorithms have been devised for processing these rules. It bears investigation what of these would work well for the Semantic Web.

Chapter 2

Background

2.1 RDF Data Model

RDF is a graph data model, wherehe nodes have labels[25]. There are labelled arcs between the nodes. An RDF *statement*, or triple, is a triple of $(subject, predicate, object)$, representing one arc of the graph. The subject and object nodes may be any of a number of things, including anonymous, or blank, nodes, literals, typed literals, and RDF symbols, uniquely represented using URIs. The predicates will be URIs.

There is a universe. Actually, there are many possible universes. Some of them are supported by the RDF graph.

Central to the idea of RDF semantics is the idea of RDF entailment. Given two RDF graphs A and B, the fragment A entails B if own can get from A to B using only the following rules

1. removing a statement
2. Take a subset $C \subset A'$, where A' is the current graph. Take a term x. Add C to the graph A' with all instances of x replaced by a new blank node y.

Two graphs are equivalent if they entail each other. A graph is *clean* if it is entailed by no strict subset of itself.

While this is the definition of RDF entailment, there may be entailments under the semantics of some RDF based languages that do not appear in the base RDF.

The RDF specification itself has some entailments not covered by the base RDF semantics. In particular, it defines a set of axiomatic triples. These triples are entailed by every RDF graph. `rdf:_n rdf:type rdf:Property` is an axiom for all integers n, so this set is not finite.

2.1.1 Modeling Data as RDF

A relational database is a set of k-ary relations, $(n_1, n_2, \ldots, n_k)$. This is easy to model as RDF. We create k predicates. Then, we have $({}_n, r1, n1)({}_n, r2, n2)\ldots({}_n, r_k, n_k)$, where ${}_n$ is an anonymous node. In this way, a k-ary relation was encoded in k ternary relations. This is often the best way to model data in RDF [1].

The W3C Semantic Web Best Practices Working Group[42] publishes recommendations about how to represent data as RDF. Many things can be represented in multiple ways, and some ways are better for some things than others.

2.2 RDF Syntaxes

The usual syntax for RDF [43] is difficult for both people and computers, though for different reasons. Throughout this thesis, we will be using the Notation3 syntax [4]. Notation3 is a syntax for RDF explained in [7]. Subsets of Notation3 which do not have extensions to RDF have been defined; in particular, there is Turtle [3].

A Notation3 statement is formed by *subject predicate object.* , terminated as here with a period. Multiple statements with the same subject, or subject object, can be separated with a semicolon or comma, respectively. IRI references are enclosed in <>. Literals are enclosed within quotes.

As a shortcut for IRI references, we can declare *prefixes.* Once a prefix is declared, it can be used in the form `prefix:localname`. There are based on Qnames in XML [10], and have inspired the CURIE standard [9].

For more about the syntax of Notation3, see appendix H

[1] It is of course possible to represent data as RDF modeled in ways that are not useful. RDF is general enough to encode anything. For instance, one can represent our data as in Appendix F. Nonetheless, data can usually be represented in ways that are reasonable.

2.3 RDFS Reasoning

RDFS adds vocabulary to RDF to talk about types. The type of types, rdfs:Class, is defined. Simple ways of describing how types as properties interact, like domain, range, and sub-class, are defined. It is very difficult to write an inconsistent RDFS document [2].

RDFS associates with each node a set known as the *class extension*. The class extension of a node y is precisely the set of terms x for which `x rdf:type y` is true. Axioms like `x rdf:subClassOf y` assert the subset relationship of the class extensions of x and y, thus the truth of statements that were never directly asserted. RDFS entailment is defined by RDF entailment of these closure graphs under the RDF semantics. Note that there are also an infinite number of axiomatic triples in RDFS, that are entailed by every graph.

2.4 OWL

This type of logical reasoning as in RDFS quickly becomes constraining. It would be nice to declare equality, and to say that things are not the members of classes. Indeed, one would like to have a very expressive logical language.

The problems one runs into with logic are inconsistency and undecidability. A naïve way of implementing a RDFS with more operators can easily run afoul of Russell's paradox[37], allowing the definition of the class of all classes that do not contain themselves. Further, it would at times not be possible to definitively answer whether an item actually belongs to a set. Indeed, allowing any sort of complex operations to define compound predicates in terms of other predicates will almost certainly lead to undecidable problems.

The solution for these problems is description logic. Description logics (DL) are subsets of first order logic, with the number of constructs allowed deliberately limited to insure decidability. The algorithms that people typically use for these are Tableau based algorithms[26].

[2] Though it is not impossible to. See Appendix A.2 for the example given in the standard

Entailment in these logics is defined in terms of consistency. If $A \wedge \sim B$ is inconsistent, then A must entail B. (If A is inconsistent, then it entails everything). The tableau can test for consistency well, therefore, it can test for entailment. [3]

OWL was designed with that in mind. It defines a vocabulary. A document written using that vocabulary may be a serialization of a DL formula of type SHOIN(D). In that case, it is called OWL-DL, and the semantics are clear. Otherwise, it is called OWL FULL, and a different set of semantics, that may lead to undecidability, apply.

We note that the entailment rules of OWL-DL are the entailment rules of the underlying description logic. Therefore, a document which is not the serialization of any description logic formula cannot be entailed by any OWL-DL graph. Because some axioms if OWL-DL are constructs built of multiple triples, simply removing a triple will make the graph invalid OWL-DL. This means that OWL-DL entailment is not a superset of even base RDF entailment, and the OWL-DL is therefore not RDF. The price we pay for representing other languages in RDF is that RDF has its own semantics and assumptions, which the representation may disallow. One of the goals of the effort for the OWL 2 standard is to fix this.

It is here that we must note that the OWL vocabulary includes terms from RDFS. Therefore, a document written using RDFS terms may have RDFS semantics, or OWL-DL semantics (OWL FULL semantics should be the same as RDFS semantics in that case). A document written with in OWL-DL will also have OWL FULL semantics, which may be different. All of these are different from the base RDF semantics. This can create headaches for RDF processors, as the RIF group has encountered [4]. Nonetheless, in general the differences are minor and making a guess is unlikely to create completely wrong conclusions with our reasoners.

In the work for this thesis, OWL-DL was not used. When OWL constructs were used, it was assumed they were OWL FULL.

---

[3]For descriptive DL's, a tableau is NEXPTIME worst case, while the problem can be solved in EXPTIME. Nonetheless, the time on problems people want to solve is far faster on a tableau than the methods which are EXPTIME.

[4]See `http://lists.w3.org/Archives/Public/public-rif-wg/2007Jul/0033.html`.. Odd things happen when we combine different RDF languages.

2.5 Rule Languages

A different subset of FOL can be used for reasoning in RDF. A Rule, where a conjunction of facts implies a fact (or conjunction thereof), is a logical construct used many places. So called expert systems do not do full FOL reasoning, but only handle such rules. Rule languages have existed for a long time, and many exist, tailored to specific tasks[39][30].

2.5.1 N3Logic

We run into the problem that RDF is not very introspective. That is, RDF graphs are not good at all at talking about RDF graphs. Considering the importance on the Web of keeping track of who says what, it would be useful to be able to refer directly to RDF graphs in RDF.

Notation3, among other things, allows this. The types of nodes allowing as subjects or objects in Notation3, besides being URI's or literals, can also be Notation3 graphs. This allows for arbitrary levels of quoting.

We note that this quoting is just that, quoting. The semantics of RDF vocabularies like OWL allow for statements to change the meaning of other statements. It is important that these not interact with the contents of quoted formulae. As the classic example, we have the superman problem:

```
:LoisLane :says {:Superman a :FlyingThing} .
:Superman = :ClarkKent .
```

It should not be true that we can infer from this that `:LoisLane :says {:ClarkKent a :FlyingThing} .`.

By adding syntax for variable declaration, Notation3 can be used to write rules. A simple N3 rule looks like

```
{?x a :Man} => {?x a :Mortal}
```

where => stands for `<http://www.w3.org/2000/10/swap/log#implies>`.

We note that it is impossible to write either NOT or OR in N3. The limitation on OR keep the reasoning steps for the rules simple. The limitation on NOT means that there is no way to express negation as failure.

The fact that there is no way to write NOT in N3 is actually a careful design decision. The world is bigger than any one reasoner knows. Just because at this time we do not know something, does not mean it is false. Reasoning where we don't know what we don't know is called open world reasoning. This leads to monotonic reasoning, which can make the rules to figure something more complicated, but gives nice properties if they ever find an answer.

By contrast, many rules languages out there use negation as failure, which can test directly for the nonexistence of a statement. These resulting programs can be unstable, with the nonexistence of a statement leading to it existing. People have devised ways of understanding such programs. Such a program can have stable model semantics[21], well-founded semantics[40][13], or completion semantics[28]. Regular Prolog has none of these, and is dependant on the ordering of the source file. All of these are subtly different. Having the programmer understand what indeed the program is doing could be difficult.

We note that the actual syntax of N3 rules does not allow for any sort of ordering or prioritization of the rules; they are unordered. Besides that, there is no real limitation on the reasoning strategy used. Indeed, several reasoners have been written. In this way, arbitrary rules from different sources can be combined in a safe way.

Built In Predicates

We note that pure pattern matching rules are often not enough. While the Notation3 language allows for arbitrary quoting levels, these by themselves are difficult to use. One would like to use them on the Web, yet nothing intrinsic to the language allows a Notation3 formula to be connected to a document IRI. Further, the Notation3 rules are limited. There are many types of operations that are simply not possible with raw rules, and would necessitate expressing the data in a level breaking way to allow

for doing those operations [5].

Towards this end, Notation3 rules have builtin predicates. A builtin predicate is defined to be a predicate that in understood natively by the reasoner. It is assumed that all triples containing the predicate that match the builtin are true. Evaluation of the builtin may involve doing arbitrary work. Indeed, the code in table 2.5.1 would create a `runAsPython` builtin, which would take the subject string, run it as a python program, and bind the object variable to what was printed.

One of these builtins in the `log:notIncludes` predicate. It tests whether the subject graph does *not* include the object graph. By doing so, it allows for tests of noninclusion without having to worry about the subtle issues of closed-world reasoning.

We note that the semantics of builtin predicates is still not well defined. Indeed, subtle changes in reasoning strategy can change whether or not they match. The basic idea is that different builtins can require either the subject, object, both, or neither to be ground terms in order to match. The `log:includes` predicate requires that the object be a N3 graph, but it may have free variables.

The implementation becomes difficult when dealing with builtins that can be functions either way. Let us look at one of them. `http://www.w3.org/2000/10/swap/math#negation` (to be referred to after this as `math:negation`) is a property between two numbers. Given a number, it is the `math:negation` of exactly one thing, and has exactly one `math:negation`. Indeed therefore, given a subject we can compute the object, and given an object we can compute the subject. Therefore, given a pattern,

```
...
?x math:negation ?y
...
```

we do not without further work know if it would match `?x` given `?y` or `?y` given `?x`. Indeed, it may depend on the facts being matched which way works.

[5]As noted in Appendix A.3, if you build your data as a giant list, then Notation3 rules are Turing-complete

```
class RunAsPython(HeavyBuiltIn, Function):
        def evaluateObject(self, subj):
                stdout = sys.stdout
                sys.stdout = newStdout = StringIO()
                exec subj in {}
                sys.stdout = stdout
                return newStdout.getvalue()
```

Table 2.1: The "RunAsPython" builtin

It may seem odd that this is the case. If nothing in a pattern can match, then it fails. Matching a variable, removing a pattern triple and variable, can never make the match harder. Therefore, the order would seem to be clear.

The problem is that builtin predicates are still pattern triples. One can add to the knowledge base the triple `1 math:negation 1 .` and it would match, despite being not being matched as a builtin predicate. Therefore, simply given a pattern it is not necessarily clear where it would start.

Further, this problem that builtins are triples too complicates the design of the matcher. Any pattern triple with a builtin predicate may match as a builtin or a pattern. Further, a pattern with a variable as the predicate may be any builtin at all. Indeed, CWM does not currently handle the last case correctly.

It may be argued that adding the statement `1 math:negation 1 .` is inconsistent, that it introduces a contradiction. As such, the reasoner may be free to ignore that statement, because the builtin returns false on it. Nonetheless, the code does not currently make that assumption, and it seems it assumes that, given the open world, it simply does not know the statement exists if the builtin returns false.

Implementing a builtin predicate was designed to be easy. Indeed, in table 2.5.1 would be the code for the `runAsPython` builtin, written for Cwm. Note that nowhere in CWM is there such a builtin. This would be a clear level breaker, and would not interact well with any other facts and rules in Notation3.

Chapter 3

Cwm Reasoner

3.1 Introduction

First written in October 2000, Cwm is a tool to explore the Semantic Web language syntax and reasoning. Cwm can read data in RDF/XML and Notation3 formats, and output in both also. It is often used simply to convert formats. It can merge documents into a larger one. Cwm understands a rule language written in Notation3, and can process those rules.

3.2 Forward Chained Reasoner

Perhaps the largest single part of Cwm is the forward chaining reasoner. Given rules written using `log:implies`[1], the forward chainer tries to match every rule, trying to find more facts. It will keep doing this until there are no more facts to be found. It is quite possible for it to loop forever instead, so it that case it will do just that. In order to do more than pattern replacement, CWM supports builtins. It is possible for a rule triple to be created in running the rule engine. In that case, the rule is added to the set of rules and processing continues.

Forward chainers are straightforward to understand. Many types of rules engines

[1] The `log` namespace stands for the URI `http://www.w3.org/2000/10/swap/log#`, so this is the URI `http://www.w3.org/2000/10/swap/log#implies`.

use them.

A problem with forward chainers is that they can easily do a tremendous amount of useless work. Given RDFS and a rule

```
{?C1 rdfs:subClassOf ?C2 .
?X a ?C1} => {?X a ?C2} .
```

It can easily figure out that everything is a `rdfs:Resource`, the universal class of RDFS. This is almost certainly a waste of work.

Of course, these rules engines can get into much worse trouble than that. Cwm allows for implying the existence of something, with the syntax:

```
{...} => {@forSome :A . ... } .
```

The reasoner must assume that $:A$, as asserted there, is different from every other individual ever seen. Therefore, it becomes trivial to assert the existence of an infinite number of statements.

```
{?X a :Man}
=>
{@forSome :A . ?X @has :father :A . :A a :Man}
```

The forward chainer will (correctly) run forever on this.

It is not even possible for any reasoning strategy to detect all cases of infinite loops. By Appendix A.3, these rules are universal. It is impossible by the halting theorem to always detect if they will loop forever. Therefore, it is likely very useful in some instances of Notation3 rules to disallow these implied existentials. So long as one does not have them (or lists, or implied nested formulae), then the number of individuals is finite, and therefore the number of possible triples is finite. Therefore, it is trivial to show that the system must halt.

If we allow implying existentials, a naïve implementation would incorrectly loop forever. If given the following (trivial by RDF semantics) rule:

```
{?X a :Man} => {@forSome :A . :A a :Man} .
```

it will loop forever creating an infinite number of statements, all saying essentially "there exists a Man." By RDF semantics, every one of those statements is redundant. Because they are not adding anything, the reasoner should detect this and stop. Cwm is not perfect in this regard, but gets most cases correct. [2]

3.2.1 Technique

CWM very aggressively indexes its triples to make search easier. There are eight types of patterns, given that there can be variables in every spot of the triple. CWM puts a triple in eight indexes, one per pattern type, so that the set that needs to be searched is greatly reduced.

3.2.2 Builtins

In order to implement builtin predicates, as described in section 2.5.1, Cwm's matcher does a large amount of work. The issue is that one could write a graph that says:

```
(1 2) math:sum 35 .
```

Cwm's matcher, at every stage, tries to find the easiest pattern triple to match. If a triple is a builtin that cannot be run yet, then by definition it cannot be the easiest pattern to match. Otherwise, the search uses the pattern with the smallest index. When it runs that, it generates all matches for that pattern, be they in the given statements or built in. For each match, it continues that search for the rest of the pattern triples.

This is a greedy *dynamic ordering*. At every step, it does whatever looks easiest right now. This may not cause the globally easiest match, but the cost of a search is dominated by the early branching factor enough that it helps.

Another thing that could be done is static ordering. By doing so, the reasoner would predetermine the order that it matches patterns to facts. The RETE algorithm[16] as described in section 4.2 does this, trading dynamic ordering for the

[2]See appendix B for an explanation of what Cwm does, and does not, do

ability to save partial matches. Now, however, matching a builtin is much harder. It must be recognized as a builtin, and put in a place low enough in the match that it can run, but high enough to be above anything that depends on it. So long as what is a builtin is clear, a topological sort would work. Nonetheless, this requires information Cwm does not have. Its builtins only answer if they can now run, not what they need to be able to run.

3.3 Backward Chained Reasoner

The essence of backward chaining is to look starting from the answer needed to find the data needed to answer it. By doing this, in many problems much less work needs to be done than with a forward chainer, because much of the data can be ignored. However, depending on the queries generated, it may very well be that a backward chainer does more work.

Backward chaining runs into problems that do not exist in forward chaining. Indeed, the naïve way of implementing a backward chaining reasoner, as Prolog does it, has severe looping problems.

As an example, we will look at a simple rule, that might appear in a ruleset for processing RDFS. The rule is

```
{?x a ?C1 . ?C1 rdfs:subClassOf ?C2} => {?x a ?C2} .
```

We note that running the Prolog rules on this rule, and a query `:Bob a ?C`, generates the query `:Bob a ?C1`. This is then run, generating the query `:Bob a ?C1`. This is then run. We see that the reasoner is caught in a loop, trying to answer a query by running the same query again.

Prolog systems solve this problem by a combination of two factors. First, they are looking for a *a* answer, not *every* answer. Therefore, once one answer is found, it stops. It does not have to get into cases that may loop trying to find other answers. The other one is that the rules are ordered. By putting non looping rules before looping ones, it will avoid loops if possible. In combination, these two allow the programmer to avoid infinite loops.

We note that neither of these properties are true in Notation3 rules. A Notation3 graph is unordered, and therefore there is no way to order the rules. Further, this lack of order means that we do not know which answer we want. In truth, we want the system to returns every answer to a query.

The idea that Euler [36] uses is loop detection. At ever step of the query, it keeps a list of previous queries in the stack. If the current one is equivalent to a previous one, the query fails right there. In our example, at the point of the first `:Bob a ?C1` query, it would detect the loop and fail, backtracking the search.

We note that there is still a great deal of redundant work possibly being done. In a single search tree, it is possible for the same query to come up again and again, so long as it is never from a descendant of another copy.

The solution I preferred was tabling. Essentially, tabling builds a table of queries and their results. When running a query, the object associated with it is registered in the table. Any queries generated are first checked in the table. If they are not there, they are run. If they are there, a listener is installed for whenever they return a result. This stops those trivial loops.

As an example, let us look at the `subClassOf` rule seen before. Starting from a query `:Bob a ?C`, an query object for that is created and added to the table. At that point, the rule shown is selected. Two more queries are needed. The first, `:Bob a ?C1`, is already in the table. Therefore, it installs the callback from that object. The other `?C1 rdfs:subClassOf ?C2`, is then instantiated. In this way, the query process terminates.

The lack of negation as failure in this logic creates some good properties for this reasoner. While [13] has a much more complex tabling algorithm to handle negation, we don't need any of it. The result is, that the algorithm is relatively straightforward.

A backward chainer is not perfect. For instance, given a rule to support `rdfs:subPropertyOf`,

```
{?P1 rdfs:subPropertyOf ?P2 .
 ?S ?P1 ?O } => {?S ?P2 ?O} .
```

It is possible for a query of `?S ?P2 ?O` to be generated. This will cause a query for

every statement to be generated, and remove all benefit from the backward chainer. The backward chainer will do all of the work of a forward chainer anyways.

It is a result of the halting theorem that no technique is perfect; there is no way to stop the reasoner from running forever on some inputs. Clearly, with a backward chainer some inputs require much less work.

The backward chaining reasoner also can compute results that the forward chainer will refuse to. One such example is `http://www.w3.org/2000/10/swap/test/include/t10.n3`, which says in part:

```
{ :doc log:semantics :F .
  :F  log:includes :G } log:implies  {  :doc local:says :G }.


{ <t10a.n3> local:says { :theSky :is :blue } }
log:implies {:test_SURPRISE a :success}.
```

The builtin system for Cwm will refuse to find the `log:sematics` of a non-specified file, so it cannot run this. However, the backward chainer will note that it wants to know if `:test_SURPRISE` is a `:success`. This leads to the query if `<t10a.n3> local:says { :theSky :is :blue }`. The query that is run for the builtin predicate is therefore `<t10a.n3> log:semantics :F`, which it can run.

In order to implement the backward chaining, the indexes needed to be extended. A statement like `@forAll :X . :X a rdfs:Resource` will match many ground queries. Because of this, any query triple will need to read from up to eight indexes instead of one. Each triple will still appear in eight indexes. For a ground triple, it will appear in the same eight. For a triple with a variable, the variable will be replaced by a generic variable placeholder before choosing indexes. Let us take an example.

While there are very few top-level statements with variables, in a backward chainer, there are many statements with variables. The rule

```
{?X a ?C1 . ?C1 rdfs:subClassOf ?C2} => {?X a ?C2}
```

generates the statement `?X a ?C2`. There are two variables. This would appear in eight indexes. These are:

VAR	rdf:type [3]	VAR
None	rdf:type	VAR
VAR	None	VAR
VAR	rdf:type	None
None	None	VAR
None	rdf:type	None
VAR	None	None
None	None	None

The corresponding query `?X a ?C2` would be querying from two index lists. It would be using

None	rdf:type	None
None	VAR	None

The variables must match `None`, because they will match anything. Because `VAR` also goes into `None`, the subject and object with `None` are redundant. A ground query will combine eight indexes.

3.4 SPARQL query engine

A recent effort was to create a query engine for RDF data and the Semantic Web. The result was SPARQL[35], a standard query language for the Semantic Web. A SPARQL query is an instruction on what to return and a patterns or expression of patterns to match in one or more RDF graphs.

It was noticed that a query engine is a piece of a rule engine. A SPARQL query looks a great deal like a Cwm rule. Adding SPARQL query support to Cwm was considered a useful feature. The question was how to implement that feature.

A typical SPARQL query may look like:

```
PREFIX : <http://example.com/#>
SELECT ?x
WHERE {?y :p1 ?x . ?x :p2 :Foo}
```

This is very similar to the Notation3 rule:

```
{?y :p1 ?x . ?x :p2 :Foo} => {(?x) a sparql:Result} .
```

This is why it was thought that the mapping would be natural.

At the time this implementation work was done (summer 2005), SPARQL was not yet a Recommendation. The section of the standard specifying how to process queries was not written yet. There was a great deal of freedom in how to implement the query engine. Given this, I tried to write a converter from SPARQL to Notation3 rules. Then the existing query engine could run the queries.

`OPTIONAL` in a sparql query allows for the optional matching of part of a graph. If the part inside of the optional section can match, then it must. Otherwise, the query can succeed without it.

It quickly became clear that the translation was not as straightforward as first thought. The use of nested `OPTIONAL` keyword blocks in a query starts to get some very unclear semantics, and does not map well to the Notation3 rules. Combinations of `OPTIONAL`s and `UNION`s were resulting in some weird interactions, that had some complex translations.

The `FILTER` expressions caused bigger problems. A `FILTER` is an algebraic expression that must be true for the match to succeed. Each function in a `FILTER` expression has input types and output types. Which Notation3 builtin was mapped to would depend sometimes on the input types. Further, there were type coercions when the output type of one did not map to the input type of another. Sometimes, the result of an expression was a TypeError. This may have come up when compiling the `FILTER` expression, or when running the query. Finally, TypeError is a value that is neither TRUE nor FALSE, and behaves oddly in Boolean logic.

The lack of support for OR in N3 rules became a problem. Splitting the query into two for every choice works when there are just `UNION`s, and a few of them. But `http://www.w3.org/2000/10/swap/test/sparql/union5.sparql` shows the exponential blowup. Further, this requires all ORs to be at the top level. The rules for distributing OPTIONALs and ANDs over ORs are in fact not always possible in the presence of

TypeError.

In order to talk about parsing, let us talk about parsers and grammars. The power of the machine needed to recognize a language is its expressivity. Typically, parsing is done by a context free grammar, modeling a nondeterministic stack machine. Given that our computer are deterministic, emulating this on a computer is not always trivial. Therefore, subsets of the class of all context free grammars are used.

A bottom up parser builds up states from the bottom up. Things tend to associate from right to left. These are known as LR(k) parsers. A top-down parser does the opposite, and is known as a LL(k) parser.

In both of these parsers, the (k) determines the number of tokens the parser remembers at once to figure out the state. The lowest numbers, and most commonly appearing, are LL(0), which saves one character, and LALR(1), which is not a full LR(1). A nondeterministic parser would not work with any finite k.

It is important to note that people do not write LR parsers. Rather, people write code that is compiled into the LR parser. The actual parser is a table

The first step in the process was the building of a grammar. The grammar provided in the SPARQL standard was not sufficient. It was designed for a LALR(1) parser, while the parser engine I had was a LL(0) parser. I figured out how to modify the grammar to have the correct expressivity. The resulting grammar is, unfortunately, much less straightforward.

Once it is done parsing, a parse tree is generated. This is converted in a bottom-up manner into the Notation3 rules. However, as we have seen, this conversion is far from being straightforward.

3.5 Proof Generation and Checking

For any sort of problem solving, there is the distinction between solution finding and solution checking. If one finds a solution, then one can go through the steps one went through to find the solution and print them out. Given these, verifying that a solution was indeed given is linear in the length of the step list (so long as each step

is constant time). This is while there may be no bound whatsoever on the amount of work needed to find a solution. Indeed, the problem may be undecidable, having no solution a computer can find, and have no decidable proof of its undecidability.

For easier problems, we have the distinction between P and NP. NP is the class of problems whose proof trace of steps to find the solution is polynomial in the length of the input. There are problems in NP that we are certain (though we cannot prove) require exponential time to find the solution in the first place.

Note that this is a slightly different idea than the general computability distinction. The checker of a general program has to do the same work as the original program. However, it knows before it starts how long it will take. By contrast, most of the work done on a problem in NP by a solver is wasted. Therefore, while it spent exponential time, the actual trace generated is only polynomial length. It is precisely these problems that are in NP. (We note that for a problem in NP, the best time cannot be worse that exponential. Indeed, a simple implementation searches for every possible proof, of which there are only an exponential number).

The general idea of these is that search is hard. It is always good to have others do the searching, and simply tell you how to go.

This should be true for Notation3 rules as well. It turns out, that finding out how a decision was made is often more important that what the result was. Further, the party interested in computing a result may not be the party interested in its correctness. Generating proof allows for these jobs to be separated.

Generating the proofs is reasonably straightforward. When a match is made, what was matched was stored. This combination of the rule, what was matched, and the result, is saved. When asked to print the proof, these are strung together.

One aspect of this strategy is that builtins must be handled separately. For most builtins, it suffices to have an object that simply knows which builtin was run. For builtins that run the reasoner recursively, like `log:supports`, the proof can include a subproof.

Towards that end, there is a proof format for Notation3. Continuing with the theme that there should be one format for both data and rules, the proof format is

also in Notation3. In this way, tools to manipulate N3 can deal with proofs as well.

Nonetheless, the proof format runs into difficulties that other uses of Notation3 do not run into. Essentially, Notation3 turns out to be bad at talking about Notation3 — we have still not reached a fully introspective language. These problems are mostly problems of denotation, including one that is also a problem of equality.

The problem of equality is mostly a problem of computational difficulty. Given two Notation3 formulae, are they equal? There are many different definitions of equality. Let us have an example. The following are four N3 formulae.

```
:A = {_:a :b :c . :c a :Q } .
:B = {:c a :Q . _:a :b :c } .
:C = {_:b :b :c . :c a :Q } .
:D = {_:a :b :c, _:e . :c a :Q } .
```

Are A, B, C, D equal? If we define equality as having the exact same serialization, then none of them are equal. If we say that reordering does not change identity, then A and B are equal. If we say that renaming variables does not change identity, then A and C are equal. Finally, if we say that if two formulae are true in the same interpretations, then D is equal to the others.

What is wrong with saying that they are all equal? The problem is that it is too limiting. We would like to be able to say

```
{@forAll :X . :X a [ owl:disjointWith :Immortal ] } # nothing is immortal
  :hasVariable :X .
```

The question is: is `:hasVariable` a valid predicate? It would be nice if it was. We would have a much easier time talking about N3 in N3, which is important in the proof format. We would like to say what a variable was bound to. In order to do that, we must be able to denote the variable from outside of its declaration. This is logically invalid. Therefore, we would like to at least allow this, and claim that there is some meaning to variable names in Notation3.

Given that, the problem of comparing two Notation3 formulae for equality is simple. Sort the two, and compare them. This is $O(n \log n)$ time, which is difficult

to improve upon.

Of course, things are much worse. There is no requirement that a node even have a name. Thus, we note the following:

```
{ @forSome :x . :x a :Thing } :hasVariable :x .
{ [ a :Thing ] } :hasVariable :x .
```

CWM will convert the first line into the second when processing a N3 file. Indeed, the "[]" notation is used to allow for just that removal of unnecessary names. Even so, how can one refer to this node that was never named? Further, testing for equality just go much harder.

If we look at the proof ontology for Notation3 [4], we note these problems show up. In order to say what variable bound to what, it explicitly says so. Variable names are represented as strings, in order to be able to refer to them unambiguously. Yet, when actually quoting the rule, the variables in the rule represent themselves.

The main purpose of a proof format is to show exactly what steps were taken. This way, a reader can follow those steps, verify their correctness, and thus verify the result.

Vagueness therefore does not support proofs. In general, a proof format should be able to say precisely what operations were done on precisely what data. Any less would be leaving too much to the reader.

Here we have a problem. In RDF, there anonymous nodes. These are very difficult to refer to unambiguously. Nodes that were never given names at all are particularly bad.

One option would be to have some standardized naming scheme based on location. Indeed, CWM does name anonymous nodes based on the line and column in the file. This is standardized nowhere, however. Further, this means that changing seemingly unimportant spacing in the RDF or Notation3 file would change the names of the nodes. Further, many RDF tools will do other transformations including reordering the file.

[4] The ontology appears at `http://www.w3.org/2000/10/swap/proof`

```
_:a0 :p _:a1 . _:a1 :p _:a0 .
_:a2 :p _:a3 . _:a3 :p _:a2 .
_:a4 :p _:a5 . _:a5 :p _:a4 .
_:a6 :p _:a7 . _:a7 :p _:a6 .
_:a0 :p _:a2 . _:a2 :p _:a0 .
_:a1 :p _:a3 . _:a3 :p _:a1 .
_:a4 :p _:a6 . _:a6 :p _:a4 .
_:a5 :p _:a7 . _:a7 :p _:a5 .
_:a0 :p _:a4 . _:a4 :p _:a0 .
_:a1 :p _:a5 . _:a5 :p _:a1 .
_:a2 :p _:a6 . _:a6 :p _:a2 .
_:a3 :p _:a7 . _:a7 :p _:a3 .
```

Table 3.1: Some very symmetric Notation3

As an example, let us look at the Notation3 in Table 3.5. We can see the cube shape of this graph in Figure 3-1. Note that all rotations of the nodes leaves everything the same. Because of the way Notation3 works, all blank nodes must be given names in the file. Nonetheless, there is no way to distinguish any of the nodes. They are all the same by rotations. If we were to compare this to another RDF file for equivalence, tools like *cant*[5] will simply refuse.

Cant [5] is an RDF canonicalizer. Given an RDF graph, it will try to find a canonical serialization for it, following techniques in [29]. If it can find canonicalizations for two files, then they are equivalent up to node renaming if and only if it gives the same canonicalizations. However, for harder cases like our cube it simply gives up. Further, it does not try to handle finding redundancy within an RDF graph.

Finding redundancy is in general really hard. To decide if a graph is *clean*, or has no redundancy, one needs to verify that there is no subset of the graph that entails the whole graph. To decide whether a particular subgraph entails the whole graph is at least as hard as graph-subgraph isomorphism, which is NP-Hard. This is worse, because a graph can be redundant in multiple ways.

We can prove that RDF matching is at worst graph-subgraph isomorphism easily. If we want to find out if undirected graph A is a subgraph of undirected graph B, in both replace each undirected edge with two edges, as in our cube. Then make an

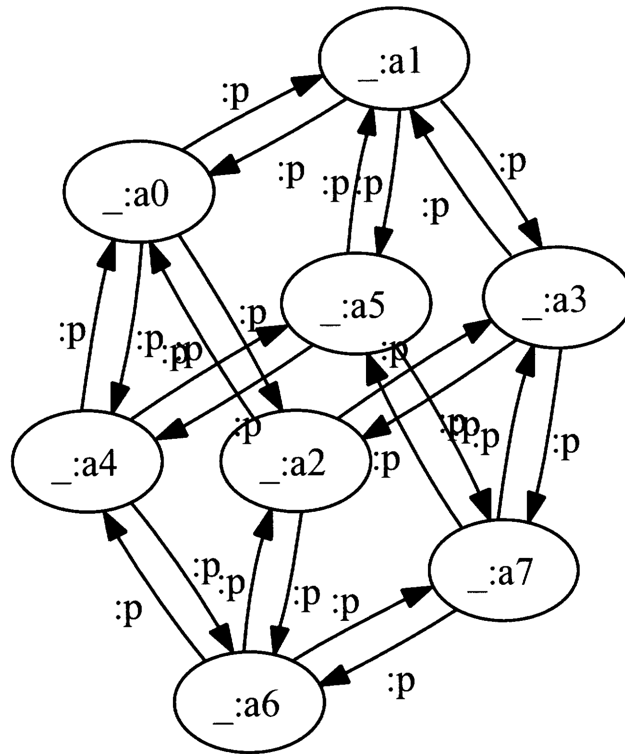


Figure 3-1: A cube of anonymous nodes

RDF graph G that is $A \cup B$. Then we will see if this is equivalent to B.

The good news is that people tend not to write RDF files that have structures like this. Indeed, almost all anonymous nodes that exist can be trivially proved to be "nailed down" by the properties that describe them in relation to named nodes. Thus, while this task is in the worst case still NP hard, in practice it is almost always much easier.

The other good news is that people tend to write clean RDF. Every once in a while a computer reasoner will make an RDF graph that is redundant, but these are usually pretty trivially redundant as well. Few operations actually create unclean RDF.

The bad news is that there is a second type of node in Notation3 files that is difficult to uniquely identify. A quoted formula is identified only by its contents, which as we just demonstrated, is difficult to identify.

Carlos Kloos has implemented an idea of hashing. By replacing every variable

quantified at a level with the same value, we can recursively create a hash value for all kinds of Notation3 nodes such that two logically identical formulae have the same hash value, so long as they are both clean. Given that, different rules will have different hash values, and be identifiable easily. Because the proof process will not be creating spurious redundant triples, this is a sufficient definition of identity for the problem. This code was never integrated into the proof checker, so it is unknown how well it would work in practice.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 4

AIR Reasoner

4.1 Introduction

When expressing a policy, one uses rules. A rule states that in certain circumstances, something applies. There are many ways of modeling this, but one of them is a rule language.

The TAMI project[22] is a project for finding ways of keeping privacy in an increasingly connected world. One of the efforts of the project is finding ways to express and enforce policy. The AIR language[23] is one such attempt.

For building a policy language, a rule language similar to Notation3 rules was thought of to be useful. However, justifications were thought to be perhaps as important as the results. As a result, several different choices were made.

The first major change is architectural. The RDF store is replaced by a *Truth Maintenance System* (TMS) [17]. The TMS provides considerable power in a very simple mechanism; its primary cost is the memory required to record the structure of a derivation. Although the TMS technology was invented in the 1970s, it is not well known outside the artificial intelligence community, and consequently there are no uses of this technology in policy systems of which we are aware.

The second major change is in syntax. In order to refer to rules in a more clear way, it is useful to name them. This makes rules take up more than one triple. Once this happens, having a rules imply another rules requires special syntax.

Indeed, a rule can have any number of `air:assert` clauses. However, unlike Notation3 rules which can imply the existence of other rules simply by having the other rule in the consequent, an AIR rule is a structure made of multiple triples. To allow for rules to be created with the data would allow for a set of triples created with separate rules to spontaneously become a rule. This would increase the complexity of the implementation. Therefore, a rule can have triples the have it as the subject, `air:rule` as the predicate, and another rule as the object. This other rule, with the correct variable bindings, becomes active with the firing of the outer rule.

A second problem is that a search can be either data directed or goal directed. However, both are wasteful. Pure goal direction, as in Prolog, will search for many facts which do not exist. Pure data direction, as in most production rule systems, will find many facts which are not interesting or wanted. To search from both ends, with goals and facts both directing the search, can be more efficient than either.

AMORD[14] is a production rule system like this. By writing rules to carefully manipulate goals, it is possible to cut down on search. As a production rule system, the rules are instructions to the system, and are not intended to have inherent logical meaning.

A typical way of programming AMORD is to have facts themselves, and have goals of those facts. One can write rules to manipulate goals, and only generate facts when needed. In this way, AMORD can act like a hybrid of a forward and backward chainer.

AMORD also shows how this approach can lead to problems. Rules to manipulate goals that are not written carefully can loop backward forever like any backward chainer. Correctly writing rules that run efficiently is a subtle art.

In AIR, the TAMI project took this goal manipulation and made it part of the language. Having a pattern in a rule leads to that pattern being a goal. A **goal rule** matches a set of triples, at least one of which is a goal. A goal rule will almost never create a fact — few things are true simply because we want them to be. Typically, it has a sub-rule which is a normal rule, and is therefore controlled by the goal.

Therefore, AIR has a syntax that names rules. Every rule has a pattern, and some

number of assertions. It can have some number of subrules, which assert a rule. It has its set of variables as a property, instead of as Notation3 variables, for historical reasons[1]. A set of rules can belong to a *policy* which is run by the reasoner. The AIR ontology has all of this, and more.

As an example, we will look at a rule to implement the `rdfs:subPropertyOf` property. If a property $P1$ is a sub-property of $P2$, then

$$\forall x, y. P1(x, y) \rightarrow P2(x, y) \tag{4.1}$$

We can encode this as an AIR rule very easily. It looks like

```
:subProperty1 a air:Rule;
     air:variables :x, y, :P1, P2;
     air:pattern {?P1 rdfs:subPropertyOf ?P2 .
                  ?X ?P1 ?Y};
     air:assert {?X ?P2 ?Y} .
```

Notice exactly how bad this rule is for AIR. The pattern `?X ?P1 ?Y` creates the most general goal. Once this goal is created, the goal direction has essentially been bypassed. Therefore, more effort must be made in writing the rule.

Simply encasing the rule in a goal rule would not work. The goal would be `?X ?P2 ?Y`, which is also too general.

The solution is to nest the rules, carefully controlling rules using goals and goals using rules. This would look like

```
:subProperty2 a air:Rule;
     air:variables :x, y, :P1, P2;
     air:pattern {?P1 rdfs:subPropertyOf ?P2};
     air:goal-rule [ air:pattern {?X ?P2 ?Y};
                     air:rule [ air:pattern {?X ?P1 ?Y};
                                air:assert {?X ?P2 ?Y} ] ] .
```

[1]The semantics for this are confusing at best. There is a reason why variables are a *syntactic* property in Notation3

This is how the rule in `http://dig.csail.mit.edu/TAMI/2007/amord/base-rules.ttl` is done.

It is not always obvious what the correct way to write a rule is. That would depend on the interactions of the rules in the system. Perhaps this is a case where an optimizer written in software would be able to do a better job than the programmer.

A Truth Maintenance System (TMS) [17] is a system to track dependencies. By doing so, it figures out what results are based on what assumptions. One can easily add a satisfiability tester to this, to find what is needed for something to be supported. One can assume and retract hypotheticals, to find out what results from various assumptions.

The main use of the TMS in AIR is to support the printing of justifications. It was deemed more interesting to know why a particular decision was made, rather than what was made. Given that the system will never have perfect information, and at best it can be advisory towards people, reasons are very important.

TMS's have been used by others in RDF. The basic idea of using a TMS to track derived facts, allowing for the removal of derived facts if the base facts they depend on are removed, is used by [11].

As part of AIR rules, one can control the generation of dependencies. In this way, the justification traces that are printed can skip rules entirely, or classes of rules deemed to be not interesting for whatever reasons. In particular, goal rules do not manipulate any actual facts, so should not appear in the trace.

The syntax for this allows for the matching of matched graphs, and the building of TMS justifications out of these.

If no dependencies are given, then the rule to be used for a rule is that everything that is created by the rule depends on an `AND` of everything that was used to fire the rule — the rule itself, and all facts it matched.

Other metadata can be given as properties to the rule. One that has been useful for human output is `air:description`, which has in its range a list of strings and RDF nodes. When the variables are substituted into it, this is put into the justification output, forming a human readable version.

In order to support a form of closed-world Negation-as-failure type reasoning, one can supply an `air:alt` property to a rule. This given what should be assumed if the rule fails to fire. This is put into the TMS with a node specifying exactly what was assumed that failed to fire the rule. In effect, it is making explicit the closed-world assumption. This is easier to work with than `log:includes`, though it runs into the problems of instability.

An example of this instability is from a file reproduced in Appendix G.

Running it through the current AIR reasoner fires exactly one of the two rules, despite them being identical. Exactly what it should do in this case has yet to be determined.

4.2 RETE AIR Reasoner

In needing to build a reasoner, several design decisions were started with. First, I was going to write in Python [41]. This is the language that I have experience programming in, and whose libraries I know. Python has its own tradeoffs, which we will deal with in section 4.7

An RDF library was needed, and Cwm was chosen. The reason for this was trivial; as Cwm's maintainer, I know that code and can modify it to suit my needs. Much of the functionality needed had been written or improved by me.

The first step of my work was to write a TMS. The TMS was based on a `tms.scm` written in scheme by Chris Hanson. Cwm was modified to use this TMS as the basis of the working store of things it currently believes.

The rule engine I wrote is an implementation of RETE-UL [16], an advance of the original RETE algorithm by Forgy [19] by Robert Doorenbos. Work was done to adapt the algorithm into Cwm.

The Cwm RDF store keeps eight indexes. These indexes correspond precisely to the alpha nodes of a RETE network. The basic idea was to do just that; make the existing indexes into alpha nodes.

What quickly became clear was that these alpha nodes did not do enough tests to

be true alpha nodes. In particular, an alpha node must do all intra statement tests, but the indexes had no way to test variable correctness. In particular, the patterns like `?x :p ?x` must be handled correctly.

The solution was to split the alpha nodes into two. The initial indexes send their data to alpha filters, which figure out the variable bindings and ensure intra-statement correctness. These used the already existing unification routines in Cwm.

In order to handle variables in facts, which must happen, variable renaming had to be implemented.

The beta network then simply tests for variable consistency, and passes on the consistent ones. In following [16], there is a root node that is the start of every merge. It has as a child some number of merge nodes. Each merge node has a left parent, which is a beta node, and a right parent, an alpha filter. When something is added to an alpha filter, it call a corresponding method on all merge nodes that are its children. Similarly, when something is added to a beta node, it calls a corresponding method on all merge nodes that are its children. When poked from one side, the merge node searches through everything on the other side, and passes on to its child beta node all consistent matches.

The alternative was to have the top beta node have two alpha nodes as parents. The alternative, however, does not handle zero- or one- triple patterns well, and is simply less consistent. Also in following [16], the beta nodes were split into beta memories, which store partial matches and have one parent, and join nodes that have two parents and no memory.

When a join node queries its parents, it would be possible to use heavy indexing on variable bindings to reduce the cost of the search. Pychinko [33] does this. The policy reasoner does not do this. By a simple indirection in the API between nodes, the API already supports this, if adding it in the future were found to be helpful. This is done by insuring that instead of searching through the alpha filter directly, it calls the **`AlphaFilter.getPossibles(bindings)`** methods to get a superset of matches consistent with the bindings. Note that the current implementation simply returns the alpha filter itself — it does nothing with the information. Further, if this method

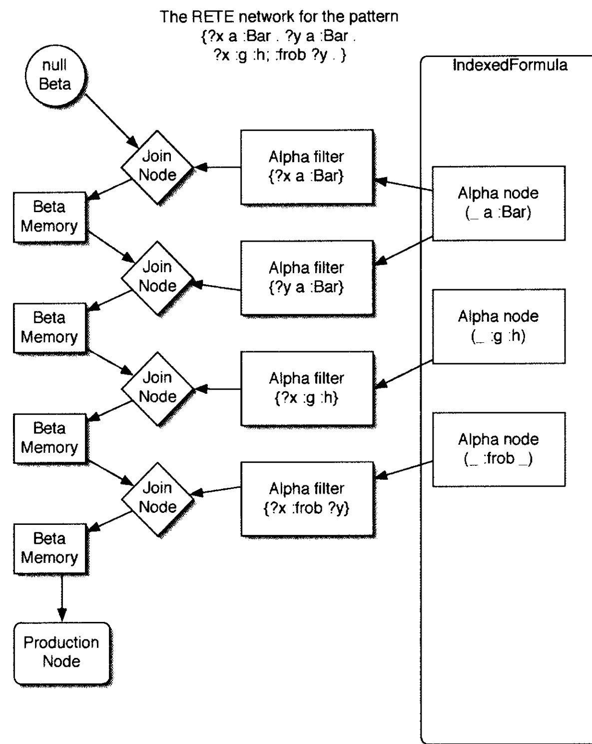


Figure 4-1: Rete Structure for a single pattern

was fully implemented, the join nodes' test would be redundant.

At the bottom of the beta node tree are production nodes. A production node has a single parent a beta node. A production node is handed a function to call on success, and a function to call on failure. In this way, the matcher knows nothing about what happens when the rule fires.

The structure 4-1 shows one matcher built this way. The pattern is

```
?x a :Bar. ?y a :Bar .
?x :g :h . ?x :frob ?y
```

This pattern has four triples, so it will be a chain of four join nodes. These are on the left. Between the join nodes are beta memories, which store partial matches. It is the job of the join nodes to only pass on consistent partial matches.

Every join node, we see, has two parents. The left parent is a beta memory, or the single root node, which acts as a beta memory. The right parent is an alpha filter. There is therefore one alpha filter per triple in the pattern.

The alpha filters get their data from the primary alpha nodes in the index of the formula. Actually, the diagram is a simplification. In the presence of variables in the facts, each alpha filter pulls data from up to eight alpha nodes, corresponding to variables in various positions in the fact. Nonetheless, we note that one primary alpha may be feeding multiple alpha filters, as both `?x a :Bar.` and `?y a :Bar.` have the same pattern to the index.

To see how this works, let us say that the TMS supports the statement `:ThreeChimneys a :Bar.`. This matches the alpha node `(_ a :Bar).` and is added. At that point, it is sent to the `?x a :Bar.` and `?y a :Bar.` alpha filters. It is consistent with both, generating the bindings `{?x: ThreeChimneys}` and `{?y: ThreeChimneys}` respectively.

The `?x` binding passes through, reaching its join node. That node merges the match with the empty one, passing it through to the beta memory. The beta memory passes it through to the next join node. This is now tested for consistency with the all of the statements in the next alpha filter. Because they have no variables in common, everything passes through to the next beta memory. In the meantime, the `?y` binding is passed through its alpha filter. It is tested for consistency at the join node with everything in the beta memory, and for the same reason, every combination is passed through.

All of those matches are then sent to the next join node. There, the bindings are tested for consistency with `?x :g :h`. Essentially, if there is a statement `:ThreeChimneys :g :h`, it is at this point passed through.

To finish the example, if anything ever adds the last beta memory, it sends whatever it had straight through to the production node. The production node then notes it is nonempty, and calls the function it was given.

As notes already, CWM support built-in predicates for RDF. To support this in a RETE, the alpha nodes must know they are built-ins and return the appropriate matches. Nonetheless, for a RETE, several difficulties present themselves.

The first difficulty is one of location. Built-in predicates typically require that either the subject, object, or both be already bound. These therefore must be placed low enough in the RETE chain to insure this true. There are some built-ins, like

`math:inverse` that given one of the subject and object, will return the other. Figuring out the correct in the presence of these is non-trivial.

The second problem is one of dynamically bound predicates. If the predicate of a pattern is a variable, it is possible that it will be bound to a built-in predicate. This is easy enough to account for, but introduces another layer of complications in any case.

Both other these problems with built-in predicates are in fact special cases of the *ordering problem*. Given a pattern with multiple statements to match, what order should be used to try to match them? The general rule is to try to fail as quickly as possible, that things which will match the least should be used. This requires dynamic ordering, to order the match on the fly as matching.

On the other hand, a RETE has a fixed structure, which determines the match order. At the time the rule is built, the order of the beta nodes, and thus the order of the match, is predetermined.

4.3 TREAT AIR Reasoner

There are several ways that RETE is faster than a naïve matching algorithm. The first is the separation of alpha nodes and beta nodes, so a pattern is only matched against statements that could match it. The second is when a new statement is added, it is automatically matched against a statement in the pattern, with the match starting from there. Depending on where in the chain the statement is added, anything from no further work to the entire rest of the match may be added. It is in the hope that the added statement will be near the bottom that the structure of the RETE is maintained. Even in the worst case, however, this is better than a naïve matcher, which would rerun the entire search for things to match to the pattern once a single statement is added.

Depending on the difficulty of a match, dynamic ordering can decrease the work of a pattern match by a great deal. It is argued in [32] that the cost of not being able to dynamically match in RETE is in fact greater than the savings from partial

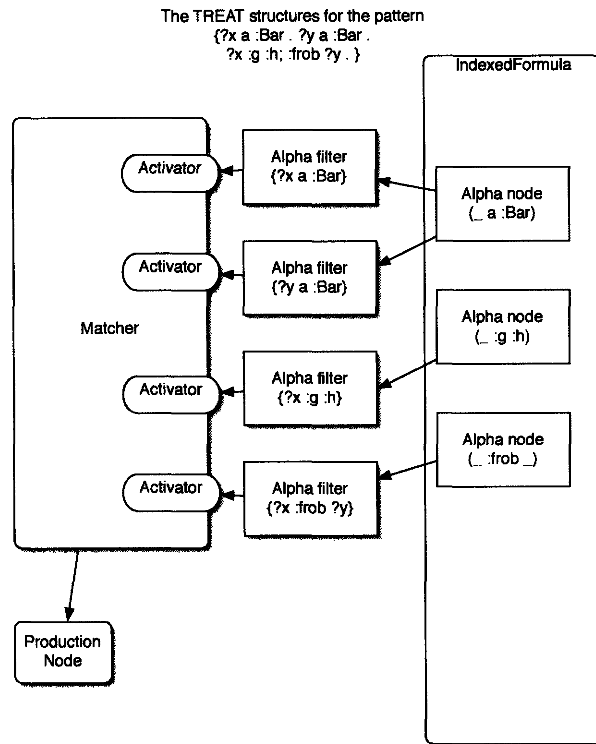


Figure 4-2: TREAT Structure for a single pattern

matches when new statements are added near the bottom.

As an alternative, they present the TREAT algorithm. The alpha nodes remain as they were. However, instead of connecting to a chain of beta nodes, the alpha nodes are all connected to a single structure representing the match. This is as in 4-2. Any time an alpha filter adds something, it pokes all child match structures. These search all other parent alpha filters for a match consistent with the new statement, and pass it on.

The advantage of having this simplified structure is flexibility. While the property remains that each statement for each line of the pattern is only matched against the possibilities for the other statements once, nothing else in forced about the process. Therefore, a dynamic order to greedily try to minimize the search space is used. This greedy ordering is found to have good properties is practice [34].

4.4 TMS

Whichever way the match happens, it calls a function given to it with two arguments. The first argument is the list of statements that were matched against. The second argument is the bindings generated. In this way, the matcher has no dependencies on any other part of the reasoner.

The function called is a method of the rule object. When called, it puts a thunk into the FIFO of things to do and returns. The thunk (a function called with no arguments) computes any assertions or subrules now supported by the firing of the rule, and adds those supports into the TMS.

We note that, in fact, so long as it was consistent with the rule we could add matches that contained unsupported statements into the TMS as well. The net result would be that the conclusion would also not be believed. The matcher is therefore simply insuring that the TMS contains what is necessary to support every true statement.

If everything worked, then the new assertion is supported. At that point, a function given to the TMS on its creation is called. This function, if given a rule node, builds the match structure for the rule. If given a statement, it adds the statement to the index for the matcher. This was, the matcher can run. The process is as in 4-3

Goal direction adds a bit of complication to this process. Goal statements can be supported in the TMS just like regular statements. Typically, a goal statement is supported simply by existence of a rule. The goals are indexed separately. Therefore, a goal rule matches against a different index. This index must include both goal statements and fact statements.

4.5 Proof Generation

The TMS is a graph structure. It contains multiple nodes for things that in presentation go together, and single nodes that should probably never be printed. It contains supports for things the user likely has no interest, and possibly support loops. It has

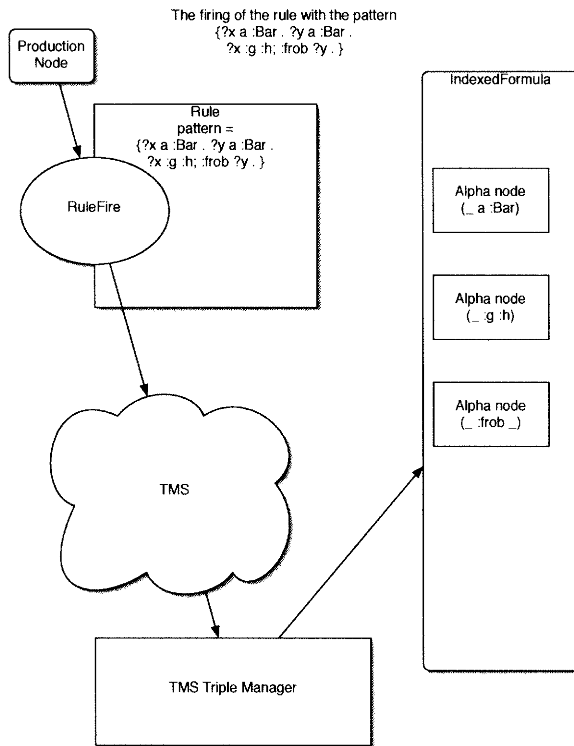


Figure 4-3: How a rule fires after a successful match

large support structures for things that are not actually supported. Therefore, effort must be taken to print out a version of the TMS that is more viewable.

The transformation is done in stages. The first stage starts with the things we want to prove, the compliance statements, and recursively makes a table of the statements used to support them. When this is done, we have a representation of the subset of the TMS needed to prove what we want. We can then transform this to make the correct output.

The first of the transformations is to remove nodes that are truly redundant. A set of statements supports all statements inside of it, and is supported by all statements inside of it. Both of these facts are not necessary.

The second transformation is more involved. If a rule was marked as being a base rule, then it should not appear in the output. This transformation cannot always be applied, and subtle effects may result.

Finally, the result is serialized as RDF. Multiple statements with the same reason

Table 4.1: The time and things generated for ARL1 and Prox card policies on sample data.

	ARL1 goals	ARL1 no goals	Prox card goals	Prox card no goals	ARL1 goals and ontologies	ARL1 no goals and ontologies
facts	121	140	125	147	589	1797
rules	319	32	395	12	810	32
RETE						
run1 (seconds)	0.80	0.46	0.97	0.50	2.40	17.69
run2 (seconds)	0.80	0.46	0.97	0.50	2.42	18.42
run3 (seconds)	0.80	0.46	0.96	0.50	2.42	18.52
minimum	0.80	0.46	0.96	0.50	2.40	17.69
TREAT						
run1 (seconds)	0.80	0.48	0.94	0.48	2.35	7.65
run2 (seconds)	0.79	0.48	0.94	0.48	2.31	7.66
run3 (seconds)	0.80	0.48	0.94	0.48	2.33	7.66
minimum	0.79	0.48	0.94	0.48	2.31	7.65

are combined for brevity.

4.6 Performance

AIR is designed to do work computing goals to avoid work computing facts.

The control structures in AIR can only do so much, however. Let us look at the behavior of the AIR reasoner on the ARL2 scenario file. The log is at `http://dig.csail.mit.edu/2008/ARL/log.n3`, and the policy is at `http://dig.csail.mit.edu/2008/ARL/udhr-policy.n3`. These are reproduced in Appendix D. The rules at `http://dig.csail.mit.edu/TAMI/2007/amord/base-rules.ttl` in appendix E are also run. These rules are trying to compute equality and RDF relations. They are very carefully written.

I also created a version that does not have any goal rules or nested rules. Any goal processing by the reasoner is wasted. It is up to the reasoner to order things. We then have figure 4.6

For the ARL1 example, running the reasoner creates 60 goal rules, 310 rules, and 75 goals. This is in order to generate 51 facts, of which we are interested in

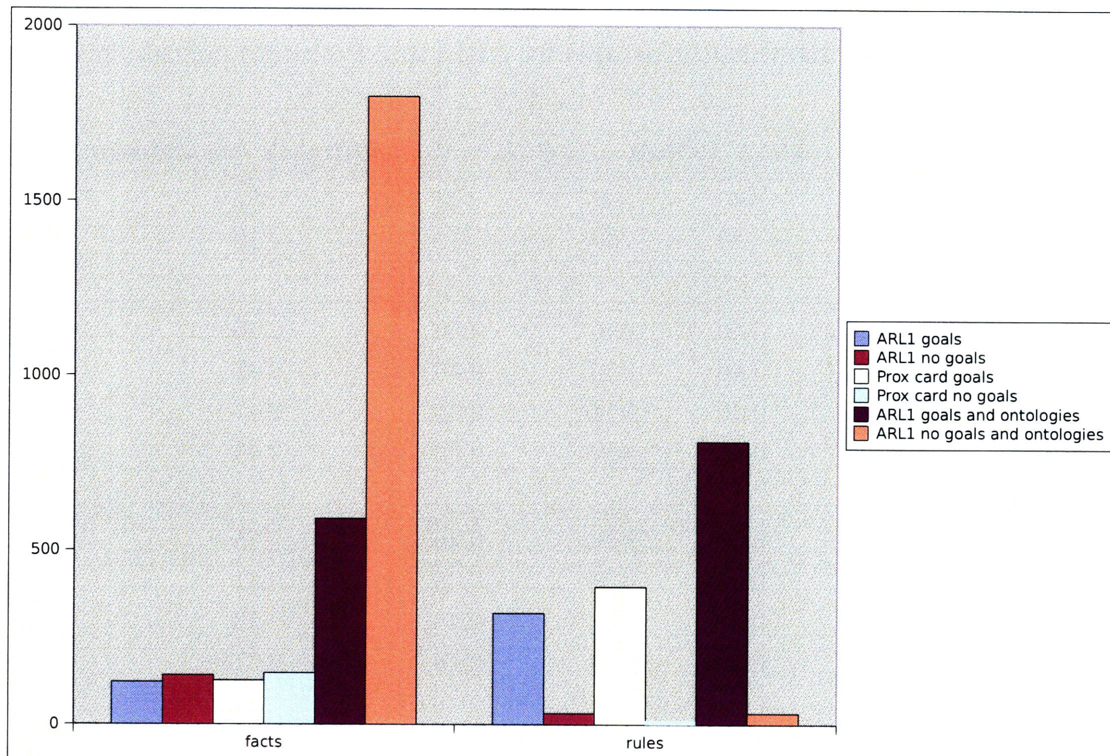


Figure 4-4: The things generated for ARL1 and Prox card policies on sample data.

less than ten. As noted in [18], we have used goals to gain some control of the generation of facts, but we don't have the same control over the generation of goals. The manipulation of goals to avoid creating facts seems to instead cause the creation of a great many rules. In this case, there were a small enough number of facts that this hurt. By loading all ontologies for the files, we add many facts, all of which end up unnecessary. The goal manipulation here works, saving work. Still, it does not prevent hundreds of facts and rules being generated.

We note that with these rules TREAT actually outperforms RETE with enough facts.

4.7 The Choice of Python

The Python[41] programming language is a dynamic typed language with everything being an object. The main implementation of Python is written in C, and a very high level virtual machine. Python's use of indentation to delineate blocks helps

force code readability. Python has a good built in library, and is in general very pleasant to program in.

The implementation of Python has some performance limitations, however. There is a very large performance penalty for interpretation. Equivalent code in C might run 1000 times faster in some instances than Python. Multithreading was added on later, with the result that there is a Global Interpreter Lock (GIL) to protect the data structures of the interpreter. This insures that a single python process cannot run on multiple processors. This can be mitigated by running multiple interpreters, one per process, and using some IPC mechanism.

THIS PAGE INTENTIONALLY LEFT BLANK

Chapter 5

Related Work

5.1 RIF

RIF is an attempt to create an interoperable rule language for the Semantic Web. It should be possible to convert a whole host of rule languages into RIF dialects, and from that into other languages. As there are many types of rule languages, not all conversions would be possible.

The Semantic Web Rule Language [27] is a proposal for a rule language to be an extension of OWL-DL. Despite that the rule language is datalog, the combination is not always finitely computable.

In particular, if a class C is defined as

```
:C owl:equivalentClass [ a owl:Restriction;
                         owl:onProperty :f
                         owl:someValuesFrom :C2 ]
```

Then the rule $C2(?X)- > C(?X)$ (encoding that $C2$ is a subset of C) will not terminate. Thus, this combination is dangerous. Nonetheless, there are a couple of implementations of SWRL.

5.2 Pellet

Pellet [38] is an OWL-DL reasoner, written to show that a sound and complete OWL-DL reasoner can be written with decent preformance. It achieves its goals using tableux based algorithms, in particular based on [26]. Pellet has grown to include support for SWRL and SPARQL queries.

The tableaux algorithms have driven the development of Pellet, which has in turn driven forward the OWL-DL standard. The sorts of queries one can ask it is defined by the OWL-DL datamodel it has. Cwm, by contrast, models RDF as RDF. To build a reasoner as complete as Pellet into Cwm would be difficult. On the other hand, it can do things that are not legal in OWL-DL.

Pellet's reasoner is complete enough to be able to do even the most difficult parts of the OWL-DL standard. For instance, they implemented a sudoku [24] ontology, which given that and a sudoku puzzle, pellet will solve the puzzle [1]

5.3 Euler

Euler is a different implementation of Notation3 and Notation3 rules. It is implemented as a backward chainer, with euler path detection to break trivial loops. It has versions written in Java and C#, as well as less complete versions in Prolog and Python. It typically generates proofs rather than simply results. Euler's parser of N3 is not complete, and given junk will happily spit out junk. Euler has no mechanism for checking the proofs it generates.

5.4 WhyNot

WhyNot [12] is a tool which is part of an inference system whose purpose is to be more ambitious than AIR currently is. While AIR rules should always generate a result, and then returns the proof for the result, WhyNot is concerned with failed

[1] really slowly. A SAT solver like minisat can solve a sudoku puzzle in a tiny fraction of the time it takes Pellet

queries in a reasoning engine. WhyNot tries to find what is likely missing from failed queries. In doing so, it tries to return what is *plausibly* the indended result, as well as why it did not work, for some definition of plausible. This is hard, and in the general case intractible, but in many cases a reasonable result can be generated.

Both WhyNot [12] and the Know system [1] focus explicitly on failed queries and try to suggest changes to the knowledge base that will cause these queries to succeed. Our justification approach is more general and allows failures to be captured in policies so explanations can be provided for both successful and failed policy decisions.

5.5 InferenceWeb

The reserchers at Stanford were interested in a general proof format. The result is IntferenceWeb [31], a format for representing inferences and proofs. This is orthogonal my work to generate and prove those inferences.

5.6 Python-DLP

The Python-DLP [15] project is interested in the intersection of DL and Horn clauses. This is the set of DL statements that can be represented with Horn rules.The FUXI project within Python-DLP has a a RETE based reasoner to process these. They are thus much more interested in DL than in rules, although by using rules to process the DL allows for combining it with other rules. They have defined a semantics for Notation3 rules [20], which mainly disallows implying existentials, which would give an expressivity greater than datalog.

5.7 Pychinko

When Cwm first came out, the reasoner was particulary inefficient at handling rules which could cause themselves to fire. In essence, what it would do is to run the rule until nothing new was generated. On a ruleset like in Appendix A.1, this would cause

it to do a tremendous amount of extra work. Also, the main indexes are generated when parsing. Only generating indexes for which there are patterns in rules could save significant amounts of time and memory. Pychinko is a Cwm clone, whose main purpose is to have better performance by using the RETE algorithm, as well as not pre-building alpha nodes. Pychinko optimizes the joining operations to a degree that none of the code I've written or maintained does. Pychinko supports a few of the simplest Cwm builtins. Testing pychinko, most of its performance advantage over Cwm on simple rulesets and small datasets is that it has faster startup time. The lazy building of indexes also contributes a great deal. It would seem to be rare for problems like Appendix A.1 to be run, where the possible algorithmic advantage shows up. Also, Pychinko never supported many builtins, seemingly for reasons that have been given in section 3.2.2.

Chapter 6

Contribution

The work done in this thesis was done over the course of maintaining Cwm as a program used by people worldwide. Cwm was originally written by Tim Berners-Lee in 2000. While doing this, I was working to ensure people could use it as a tool for real-world problems they had encountered. This work has advanced much the use and understanding of rules and proof checking on the Semantic Web.

In this work, I spent a large amount of time on rules engines. Implementing rules engines and seeing how they worked on the data that existed was an important line of work. In the process, I rewrote many parts of the reasoning engine, including the unifier and the rule scheduling.

Much of this work was done maintaining CWM over four years. Cwm was originally written by Tim Berners-Lee in 2000. The Notation3 parser was written by Dan Connolly. The basic reasoning engine was written by Tim. Various builtins were written by others, including Sean B. Palmer.

The TMS used in the AIR reasoner was translated by me, almost line for line, from scheme code by Chris Hanson. The purpose was to see what it could be used for in an RDF setting.

The AIR reasoner was written through much consultation with Chris Hanson. The design was based on an AMORD implementation he had written. The RETE engine was based on the PhD paper referenced. Modifications to support goals, builtins, and possible efficiency improvements were made.

The proof generator, and checker in Cwm were written by Tim. The checker as written by Tim was incomplete, and did not do the steps necessary to finish checking. I worked to get the proof checker both correct and performing in a satisfactory manner.

With the development of SPARQL, it was becoming obvious that a RDF engine should be able to output SPARQL. The SPARQL to Notation3 compiler I developed for this is unique. Mostly it demonstrates that the mapping between the two languages is not straightforward.

The parser this SPARQL compiler used was based on a LL(0) parser written by Tim. The grammar was processed from the standard by a program written by Eric Prud'hommeax and further hand modified by me.

By combining these different technologies into one tool, I allowed for them to be used in engineering domains that have not been explored much before. The conclusions of this thesis result from the experiences in using these tools. In particular, these experiences were resulted from running rules created by other people, and seeing if and how they could be used.

Chapter 7

Conclusion and Lessons Learned

Over the course of this work, a number of conclusions were reached. Many more things were inconclusive, or point to further work that needs to be done. Mixing rules from different sources was found to have implications on both the language and the reasoners used. There is further work to be done with reasoning strategies and proofs. The styles of rules encountered have other lessons.

A big effort by Cwm is to treat rules as data. This allows for one to talk about rules, and possibly reason over them. Naming rules, as in AIR, seems to make this easier in some senses, by making it clearer the identity of the rule. On the other hand, making a rule take up more that one RDF statement has a cost in simplicity of implementation. When treating rules as data, it become possible for a rule to suddenly appear.

In regard to choices of reasoning strategy, there are no global maxima. There exist rules for which nothing will run well, and trivial things everything will run well. In between, there is a range of problems, datasets, and rule sets. Different rule sets will be optimized for different rule techniques. In particular, some will work better with a forward chainer, others a backward chainer. It may bear investigation whether a reasoner that can make good guesses about which technique to use could be written. However, any attempts at this may well run into the halting theorem.

Rule sets vary a great deal. If one is not careful, one can write one that can loop forever. By the halting theorem, there exists a rule set that cannot be proved whether

or not it loops forever on particular data. One can write a rules set that is highly inefficient on any reasoner[1]. Given the different behaviors of various algorithms, there are many rule sets that are optimized for a particular style of reasoning. Despite the general meaning of a rule, in practice authors write rules for a particular engine.

Rules therefore must be handled with care. Any rule engine looking at the web for rules to run will run into rules that are highly inefficient on it. Putting resource limits on the reasoner in these cases would be appropriate. Looking into this would be interesting.

Another thing to look into would be further distribution and multithreading of rule reasoning. Given that Notation3 rules are monotonic, any split of the reasoning task is unlikely to be harmful.

Debugging when rules fail to get a result is a pain, and any rule based system should have something to help do this.

In regard to AIR, more work is needed for its main features. Not enough is understood about the behavior of AIR rules when processed. For instance, for complete correctness, a goal should match a goal rule if they unify at all. In the reasoner, for there to be acceptable performance, there needs to be full entailment.

This difficult to understand behavior is especially bad for the person writing the rules. Exactly how a rule is written can have very large effects on the efficiency of its processing. If techniques like AIR are to become useful, programmer effort must be decreased.

In regard to the explicit goal tracking, things are mixed. There are gains there, but they are not for all cases. This is a problem of scalability — it can help if the facts vastly outnumber the rules. Some of the gain from these rules comes from careful ordering of the match. If one does a dynamic ordering, then this is done more intelligently with much less work on the part of the programmer.

The AIR `air:alt` feature, to allow for closed-world reasoning, needs more work. Without a clear semantics, the writer of the reasoning engine and the reader of the justification trace are both likely to be confused. Unlike `log:notIncludes`, this

[1]see Appendix C

solution is not straightforward. That said, it accomplishes its goal of making certain classes of statements much easier to make.

In regard to a proof language for RDF rules, more work needs to be done. Even naming rules like AIR does should help. Hashing to identify formulae would also help. Having a way of uniquely referring to triples or anonymous nodes within the proof language would be even better. This proof language would then not be RDF based, but something else.

The essential problem was with ordering. Matching unordered things in the presence of anonymous nodes is hard. While this is rarely an issue in regular RDF, in proofs these can show up a lot. Notation3 has no good way to name anonymous nodes, for obvious reasons.

Proofs generated by computers are, in general, caught in many little details. Readability is at best hard. The work on AIR justifications generates things that would be even harder to check. Nonetheless, this work has succeeded at improving readability.

More work is needed for an easily checked proof language. Such a language would represent RDF in an unambiguous fashion, to be able to represent each proof step precisely. Given that, checking a proof takes linear time in the length of the proof. This development would be important for the usefulness of proofs on the Web. Given that information on the Web is often not trustworthy, knowing why a conclusion is reached is often as important as the conclusion, having a good system for storing and transmitting these is useful.

In summary, rules can be an effective tool on the Semantic Web. Perhaps with a better proof system, it would be possible to refer to rules and data and the conclusions based on them in a general way. In many ways, rules are too powerful to not be used, but even our current best, still often difficult to use.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix A

Example Notation3 Files

A.1 TREAT Example

Some Notation3 rules performance problems can be more easily solved than they ones in Appendix A.3 or Appendix C. As an example, we have `http://www.w3.org/2000/10/swap/test/reason/longChain.n3`

`http://www.w3.org/2000/10/swap/test/reason/shortChain.n3` is a somewhat shorter version of the same thing, which is reproduced below.

```
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#>.
 :foo a :Class1 .
 {?x a ?C1 . ?C1 rdfs:subClassOf ?C2} => {?x a ?C2} .
 :Class1 rdfs:subClassOf :Class2 .
:Class2 rdfs:subClassOf :Class3 .
:Class3 rdfs:subClassOf :Class4 .
:Class4 rdfs:subClassOf :Class5 .
:Class5 rdfs:subClassOf :Class6 .
:Class6 rdfs:subClassOf :Class7 .
:Class7 rdfs:subClassOf :Class8 .
:Class8 rdfs:subClassOf :Class9 .
:Class9 rdfs:subClassOf :Class10 .
```

```
:Class10 rdfs:subClassOf :Class11 .
```

A.2 Inconsistent RDFS

It is possible for a RDFS file to be inconsistent. The example given in the standard follows.

```
<ex:a> <ex:p> "<notLegalXML"^^rdf:XMLLiteral .
<ex:p> rdfs:range rdf:XMLLiteral .
```

A.3 Turing Completeness of Notation3 Rules

Because Notation3 does not have functions, it is not trivial that Notation3 rules are universal. Nonetheless, any implementation that allows for implying existentials, in the pattern of

```
{...} => {@forSome :A . ... } .
```

should still be universal. Included here is a file `http://www.w3.org/2000/10/swap/turing/unlambda.n3`, which is an implementation of the s and k combinators in Notation3 rules. If one accepts that this file is valid and runs, then Notation3 rules can emulate s and k combinators. The file is a simple eval/apply loop, based directly on [2]

```
@prefix : <http://yosi.us/unlambda#> .
#@prefix log: <http://www.w3.org/2000/10/swap/log#> .
@keywords a, of, is .
#
# An combinators interpreter.n3
# usage: cwm un_expr.n3 --think=unlambda.n3 --filter=un_filter.n3
# see un_expr.n3 for examples

#
```

```
# Begin defining what needs to be eval'ed
{?x a RootExpression} => {?x a NeedsEval} .

#recursive step of eval #1
{(?x ?y) a NeedsEval}
=>
{?x a NeedsEval .
 ?y a NeedsEval} .

#{?x a NeedsEval; log:rawType log:Other} => {?x eval ?x} .
#base case of eval
{k a NeedsEval} => {k eval k} .
{s a NeedsEval} => {s eval s} .
{v a NeedsEval} => {v eval v} .
{i a NeedsEval} => {i eval i} .

#
# Eval calls apply
{ (?x ?y) a NeedsEval .
 ?x eval ?a .
 ?y eval ?b .
}
=>
{(?a ?b) a NeedsApply} .

#
#Eval gets the return value from apply, and returns it
{(?x ?y) a NeedsEval .
 ?x eval ?a .
 ?y eval ?b .
```

```
 (?a ?b) apply ?c . }
=>
{(?x ?y) eval ?c } .

#
#apply (k x)
{(k ?x) apply (k ?x)}
<=
{(k ?x) a NeedsApply} .



#apply ((k y) x) ==> y
#
{((k ?y) ?x) apply ?y}
<=
{((k ?y) ?x) a NeedsApply} .

# apply (s x)
#
{(s ?x) apply (s ?x)}
<=
{(s ?x) a NeedsApply} .

# apply ((s y) x)
#
{((s ?y) ?x) apply ((s ?y) ?x)}
<=
{((s ?y) ?x) a NeedsApply } .

# apply (((s z) y) x)
```

```
# calls (z x) and (y x)
{(((s ?y) ?z) ?x) a NeedsApply .
} =>
{(?y ?x) a NeedsEval .
 (?z ?x) a NeedsEval } .

# apply (((s z) y) x)
# gets (z x) and (y x)
# and calls ((z x) (y x))
{(((s ?y) ?z) ?x) a NeedsApply .
 (?y ?x) eval ?r .
 (?z ?x) eval ?s .
}
=>
{(?r ?s) a NeedsEval} .
# apply (((s z) y) x)
# gets ((z x) (y x))
# and returns it
{(((s ?y) ?z) ?x) a NeedsApply .
 (?y ?x) eval ?r .
 (?z ?x) eval ?s .
 (?r ?s) eval ?w }
=>
{(((s ?y) ?z) ?x) apply ?w} .

# (i x) ==> x
#
{(i ?x) apply ?x}
<=
{(i ?x) a NeedsApply} .
```

```
# (v x) ==> v
#
{(v ?x) apply v}
<=
{(v ?x) a NeedsApply} .
```

Appendix B

Avoiding Redundant Rule Firings

Cwm tries to insure that it does not redundantly fire rules by treating existentials as skolem functions, functions of all variables bound that appear in the conclusion. Let us see an example.

```
{?X @has :uncle ?Y}
=>
{@forSome :A . ?A @has :brother ?Y} .
```

This rules says that an uncle is somebody's brother. We note that $?Y$ appears in the conclusion, but $?X$ does not. This does matter. If two people both have $?Y$ as an uncle, we still only know that $?Y$ is somebody's brother. If X has two uncles, they may very well not be brothers of the same person, we cannot assert `[:brother :Q1, :Q2]`. By making the existential a function of Y, we get the effect we want, at the cost of a hash table lookup.

This can still lead to the creation of redundant statements. The rule we already looked at

```
{?X a :Man} => {[a :Man]} .
```

should never fire. What it generates is redundant. Nonetheless, Cwm will generate a single `[a :Man]` triple by these rules, though it will not generate any others. In a conversation, Chris Hanson mentioned that if instead Cwm tested if the new triples

were entailed by the existing graph, given that all anonymous nodes that appear in the graph are treated as unique individuals instead of the usual treatment of blank nodes, then this would be a complete test to insure rules generate only exactly what is needed. This would involve doing a second search after the pattern in the rules already matched — a much more expensive operation. Therefore, Cwm does not do this anymore.

Appendix C

Notation3 Rules are still Computationally Intractable

The following Notation3 rule appears nowhere in any file in the wild. This is certainly because people are interested in computing useful things, and not simply making rule engines work.

```
{?X1 ?X2 ?X3 .
 ?Y1 ?Y2 ?Y3 .
 ?Z1 ?Z2 ?Z3 } => {?X2 ?Y3 ?Z1} .
```

This rule does not imply an existential, so it must terminate. That said, it has a number of bad properties.

- It matches against every statement in the system three times. This means that the number of matches generated is the cube of the current number of statements

- It generates a maximal graph. Given n RDF terms, there are n^3 possible statements it can generate. This rule will not stop until it generates them all.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix D

An example AIR file

What follows is the file `http://dig.csail.mit.edu/2008/ARL/udhr-policy.n3`. This is a simple AIR file.

```
# ARL scenario, violation of Universal Declaration of Human rights, Article 12
# http://www.unhchr.ch/udhr/lang/eng.htm

# more about ARL scenario http://dig.csail.mit.edu/2008/ARL

# $Date: 2008-02-29 18:02:15 -0500 (Fri, 29 Feb 2008) $
# $Revision: 15842 $
# $Author: lkagal $

@prefix rdf:  <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix foaf: <http://xmlns.com/foaf/0.1/> .
@prefix owl:  <http://www.w3.org/2002/07/owl#> .

@prefix air: <http://dig.csail.mit.edu/TAMI/2007/amord/air#> .
@prefix tami: <http://dig.csail.mit.edu/TAMI/2007/tami#> .
```

```
@prefix :  <http://dig.csail.mit.edu/2008/ARL/policy#> .

:Universal_Declaration_of_Human_Rights_Article12 a air:Policy;
rdfs:label "Universal Declaration of Human Rights, Article 12";
rdfs:comment "Under the Universal Declaration of Human Rights no one should be subje
air:variable :COMPLAINT, :SEARCHEVENT, :ACTOR, :ADDRESS, :DEFENDANT;
air:variable :DATA, :PURPOSE, :SPURPOSE, :PEVENT;
air:rule :UDHR_1.

:UDHR_1 a air:BeliefRule;
        air:label "Universal Declaration of Human Rights, Article 12 #1";
        air:pattern {
          :COMPLAINT a tami:PrivacyViolationComplaint;
tami:prevEvent :SEARCHEVENT;
tami:actor :ACTOR;
tami:defendant :DEFENDANT.
           :SEARCHEVENT a tami:SearchEvent.
   :DEFENDANT a tami:Military.
        };
       air:description (:COMPLAINT " was filed by " :ACTOR " about " :SEARCHEVENT " p
air:rule :UDHR_2.

:UDHR_2 a air:BeliefRule;
air:label "Universal Declaration of Human Rights, Article 12 #2";
air:pattern {
            :SEARCHEVENT tami:data :DATA;
tami:purpose :SPURPOSE.
    :DATA tami:purpose :PURPOSE.
};
air:description("Purpose of " :SEARCHEVENT " was " :SPURPOSE " and it used data meant
```

```
air:rule :UDHR_3.

:UDHR_3 a air:BeliefRule;
        air:label "Universal Declaration of Human rights, Article 12 #4";
        air:pattern {
            :SPURPOSE owl:sameAs :PURPOSE
        };
        air:description ("The purpose of the SearchEvent is the same as the purpose
        air:assert { :SEARCHEVENT air:compliant-with :Universal_Declaration_of_Human_
air:alt [ air:rule :UDHR_4 ].

:UDHR_4 a air:BeliefRule;
air:pattern {
        :SEARCHEVENT tami:prevEvent :PEVENT.
:PEVENT a tami:Consent; tami:consenter :ACTOR.
        };
        air:description ("As usage is inconsistent with purpose, consent " :PEVENT "
        air:assert { :SEARCHEVENT air:compliant-with :Universal_Declaration_of_Human_
air:alt [ air:description ("No consent was obtained by " :DEFENDANT " before " :SEARC
                  air:rule :UDHR_5 ].

:UDHR_5 a air:BeliefRule;
air:pattern {
         :SEARCHEVENT tami:prevEvent :PEVENT.
:PEVENT a tami:Notice; tami:actor :DEFENDANT; tami:notified :ACTOR.
        };
        air:description ("As usage of data was inconsistent with purpose, notice " :P
air:assert { :SEARCHEVENT air:compliant-with :Universal_Declaration_of_Human_Rights_A
air:alt [ air:description ( "No notice was given by " :DEFENDANT " before " :SEARCHEV
  air:rule :UDHR_6 ].
```

```
:UDHR_6 a air:BeliefRule;
air:pattern { };
air:assert { :SEARCHEVENT air:non-compliant-with :Universal_Declaration_of_Human_Rigl
        air:description (:SEARCHEVENT " was performed without due process, making it

# transitive closure of prevEvent
:trans-prevEvent a air:Policy;
    air:variable :E1, :E2, :E3;
    air:rule [
        air:label "Transitive closure of prevEvent";
        air:pattern { :E1 tami:prevEvent :E2.
                      :E2 tami:prevEvent :E3.
                     };
        air:assert { :E1 tami:prevEvent :E3 }
    ].

#ends
```

An example event log it can run on appears at `http://dig.csail.mit.edu/2008/ARL/log.n3`.

Appendix E

base-rules.ttl

This file was written by Chris Hanson, collaborating with Gerry Sussman.

```
# $Id: base-rules.ttl 8172 2007-12-13 19:01:57Z cph $
#
# Copyright (C) 2007 Massachusetts Institute of Technology
#
# This program is free software; you can redistribute it and/or modify
# it under the terms of the GNU General Public License as published by
# the Free Software Foundation; either version 2 of the License, or
# (at your option) any later version.
#
# This program is distributed in the hope that it will be useful, but
# WITHOUT ANY WARRANTY; without even the implied warranty of
# MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the GNU
# General Public License for more details.
#
# You should have received a copy of the GNU General Public License
# along with this program; if not, write to the Free Software
# Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA
```

```
# 02110-1301, USA.

# AIR base rules

@prefix : <http://dig.csail.mit.edu/TAMI/2007/amord/base-rules#> .
@prefix abr: <http://dig.csail.mit.edu/TAMI/2007/amord/base-rules#> .

@prefix air: <http://dig.csail.mit.edu/TAMI/2007/amord/air#> .
@prefix rdfs: <http://www.w3.org/2000/01/rdf-schema#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .

air:base-rules a air:Policy;
    air:rule
:transitive-property-implication,
:symmetric-property-implication,
:sub-class-implication,
:sub-property-implication,
:same-as-implication,
:domain-implication,
:range-implication.

:transitive-property-implication a air:Hidden-rule;
    air:variable :P;
    air:pattern { :P a owl:TransitiveProperty. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :N1, :N2;
air:pattern { :N1 :P :N2. };
air:rule [
    air:variable :N3;
```

```
    air:pattern { :N1 :P :N3. };
    air:matched-graph :G2;
    air:rule [
air:variable :N4;
air:pattern { :N3 :P :N4. };
air:matched-graph :G3;
air:assertion [
    air:statement { :N1 :P :N4. };
    air:justification [
air:rule-id :transitive-property-implication;
air:antecedent :G1, :G2, :G3;
    ];
];
    ];
];
air:rule [
    air:variable :N3;
    air:pattern { :N3 :P :N2. };
    air:matched-graph :G2;
    air:rule [
air:variable :N4;
air:pattern { :N4 :P :N3. };
air:matched-graph :G3;
air:assertion [
    air:statement { :N4 :P :N2. };
    air:justification [
air:rule-id :transitive-property-implication;
air:antecedent :G1, :G2, :G3;
    ];
];
```

```
    ];
];
    ].

:symmetric-property-implication a air:Hidden-rule;
    air:variable :P;
    air:pattern { :P a owl:SymmetricProperty. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :N1, :N2;
air:pattern { :N1 :P :N2. };
air:rule [
    air:pattern { :N2 :P :N1. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :N1 :P :N2. };
air:justification [
    air:rule-id :symmetric-property-implication;
    air:antecedent :G1, :G2;
];
    ];
];
    ].

:sub-class-implication a air:Hidden-rule;
    air:variable :C1, :C2;
    air:pattern { :C1 rdfs:subClassOf :C2. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :N;
```

```
air:pattern { :N a :C2. };
air:rule [
    air:pattern { :N a :C1. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :N a :C2. };
air:justification [
    air:rule-id :sub-class-implication;
    air:antecedent :G1, :G2;
];
    ];
];
    ].

:sub-property-implication a air:Hidden-rule;
    air:variable :P1, :P2;
    air:pattern { :P1 rdfs:subPropertyOf :P2. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :N1, :N2;
air:pattern { :N1 :P2 :N2. };
air:rule [
    air:pattern { :N1 :P1 :N2. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :N1 :P2 :N2. };
air:justification [
    air:rule-id :sub-property-implication;
    air:antecedent :G1, :G2;
];
```

```
    ];
];
    ].

:same-as-implication a air:Hidden-rule;
    air:variable :N1, :N2;
    air:pattern { :N1 owl:sameAs :N2. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :P, :O;
air:pattern { :N1 :P :O. };
air:rule [
    air:pattern { :N2 :P :O. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :N1 :P :O. };
air:justification [
    air:rule-id :same-as-implication;
    air:antecedent :G1, :G2;
];
    ];
];
    ];
    air:goal-rule [
air:variable :S, :O;
air:pattern { :S :N1 :O. };
air:rule [
    air:pattern { :S :N2 :O. };
    air:matched-graph :G2;
    air:assertion [
```

```
air:statement { :S :N1 :O. };
air:justification [
    air:rule-id :same-as-implication;
    air:antecedent :G1, :G2;
];
    ];
];
    ];
    air:goal-rule [
air:variable :S, :P;
air:pattern { :S :P :N1. };
air:rule [
    air:pattern { :S :P :N2. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :S :P :N1. };
air:justification [
    air:rule-id :same-as-implication;
    air:antecedent :G1, :G2;
];
    ];
];
    ].

:domain-implication a air:Hidden-rule;
    air:variable :P, :C;
    air:pattern { :P rdfs:domain :C. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :N1;
```

```
air:pattern { :N1 a :C. };
air:rule [
    air:variable :N2;
    air:pattern { :N1 :P :N2. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :N1 a :C. };
air:justification [
    air:rule-id :domain-implication;
    air:antecedent :G1, :G2;
];
    ];
];
    ].

:range-implication a air:Hidden-rule;
    air:variable :P, :C;
    air:pattern { :P rdfs:range :C. };
    air:matched-graph :G1;
    air:goal-rule [
air:variable :N1;
air:pattern { :N1 a :C. };
air:rule [
    air:variable :N2;
    air:pattern { :N2 :P :N1. };
    air:matched-graph :G2;
    air:assertion [
air:statement { :N1 a :C. };
air:justification [
    air:rule-id :range-implication;
```

```
    air:antecedent :G1, :G2;
];
    ];
];
    ].
```

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix F

Useless Encoding as RDF

RDF is general enough to encode data in it that is not RDF at all. The following is a valid RDF document.

```
:content = """
{\rtf1\mac\ansicpg10000\cocoartf824\cocoasubrtf440
{\fonttbl\f0\fswiss\fcharset77 Helvetica;}
{\colortbl;\red255\green255\blue255;}
\margl1440\margr1440\vieww9000\viewh8400\viewkind0
\pard\tx720\tx1440\tx2160\tx2880\tx3600\tx4320\tx5040\tx5760\tx6480\tx7200\tx792
0\tx8640\ql\qnatural\pardirnatural

\f0\fs24 \cf0 Hello}
"""^^foo:rtfDocument .
```

This is not actually encoding our data as RDF, merely quoting it in another format. Further, we can even get rid of the strange literal, making a document that looks like

```
:content = (123, 92, 114, 116, 102, 49, 92, 109, 97, 99, 92, 97, 110, 115, 105,
 99, 112, 103, 49, 48, 48, 48, 48, 92, 99, 111, 99, 111, 97, 114, 116, 102, 56,
 50, 52, 92, 99, 111, 99, 111, 97, 115, 117, 98, 114, 116, 102, 52, 52, 48, 10,
 123, 92, 102, 111, 110, 116, 116, 98, 108, 92, 102, 48, 92, 102, 115, 119,
 105, 115, 115, 92, 102, 99, 104, 97, 114, 115, 101, 116, 55, 55, 32, 72, 101,
```

```
108, 118, 101, 116, 105, 99, 97, 59, 125, 10, 123, 92, 99, 111, 108, 111, 114,
116, 98, 108, 59, 92, 114, 101, 100, 50, 53, 53, 92, 103, 114, 101, 101, 110,
50, 53, 53, 92, 98, 108, 117, 101, 50, 53, 53, 59, 125, 10, 92, 109, 97, 114,
103, 108, 49, 52, 52, 48, 92, 109, 97, 114, 103, 114, 49, 52, 52, 48, 92, 118,
105, 101, 119, 119, 57, 48, 48, 48, 92, 118, 105, 101, 119, 104, 56, 52, 48,
48, 92, 118, 105, 101, 119, 107, 105, 110, 100, 48, 10, 92, 112, 97, 114, 100,
92, 116, 120, 55, 50, 48, 92, 116, 120, 49, 52, 52, 48, 92, 116, 120, 50, 49,
54, 48, 92, 116, 120, 50, 56, 56, 48, 92, 116, 120, 51, 54, 48, 48, 92, 116,
120, 52, 51, 50, 48, 92, 116, 120, 53, 48, 52, 48, 92, 116, 120, 53, 55, 54,
48, 92, 116, 120, 54, 52, 56, 48, 92, 116, 120, 55, 50, 48, 48, 92, 116, 120,
55, 57, 50, 48, 92, 116, 120, 56, 54, 52, 48, 92, 113, 108, 92, 113, 110, 97,
116, 117, 114, 97, 108, 92, 112, 97, 114, 100, 105, 114, 110, 97, 116, 117,
114, 97, 108, 10, 10, 92, 102, 48, 92, 102, 115, 50, 52, 32, 92, 99, 102, 48,
32, 72, 101, 108, 108, 111, 125) .
```

Appendix G

The Semantics of Alt

The following file is a reproduction of `http://dig.csail.mit.edu/TAMI/2007/cwmrete/unstable.n3`. It models

$$\neg P \rightarrow Q$$
$$\neg Q \rightarrow P$$

This is a subject of the many different semantics that have been defined for Prolog like languages.

```
#### Namespaces ####

# The default namespace is this document.
@prefix : <#> .

# AIR (AMORD in RDF) is the policy language.
@prefix air: <http://dig.csail.mit.edu/TAMI/2007/amord/air#> .

@prefix owl: <http://www.w3.org/2002/07/owl#>.

#### Policy ####
```

```
:MITProxCardPolicy a air:Policy;
    air:variable :S, :P, :O;
    air:rule [
air:label "Rule 1";
air:pattern {
    :this :is :rule1
};
        air:alt [
    air:description ("It would seem that rule 1 failed to fire");
            air:assert { :rule2 :is :cool . :rule1 air:non-compliant-with owl:Thing ]
    ];
    ],
    [
air:label "Rule 2";
air:pattern {
   :rule2 :is :cool
};
        air:alt [
    air:description ("It would seem that rule 2 failed to fire");
            air:assert { :this :is :rule1  . :rule2 air:non-compliant-with owl:Thing]
    ];
    ] .
```

After running the command `python policyrunner.py test file:‘pwd‘/unstable.n3 file:‘pw` the following output was created.

```
 @prefix : <http://dig.csail.mit.edu/TAMI/2007/amord/tms#> .
@prefix air: <http://dig.csail.mit.edu/TAMI/2007/amord/air#> .
@prefix owl: <http://www.w3.org/2002/07/owl#> .
@prefix prox: <file:/home/syosi/classes/meng-thesis/inst.n3#> .
```

```
prox:rule1     air:non-compliant-with owl:Thing .

<#_g0>     :justification :premise .
{

    }     :justification :premise .
{
    prox:rule1     air:non-compliant-with owl:Thing .

    }     :description  (
    "It would seem that rule 1 failed to fire" );
     :justification  [
         :antecedent-expr  [
             a :And-justification;
             :sub-expr <#_g0>,
                     [
                 air:closed-world-assumption  (
                prox:MITProxCardPolicy
                air:base-rules
                <file:/home/syosi/classes/meng-thesis/inst.n3>
                <http://dig.csail.mit.edu/TAMI/2007/amord/base-assumptions.ttl> )
                 :justification :premise ],
                    {} ];
         :rule-name <#_g0> ] .
```

This shows exactly one of the two identical rules fired, which is unstable.

THIS PAGE INTENTIONALLY LEFT BLANK

Appendix H

The Notation3 Syntax

Notation3, or N3, was a syntax for RDF that evolved from a syntax used on whiteboards. Its main purpose is to be more readable, writable, and extensible than RDF/XML. The full grammar of Notation3 is at [4] and [6]. What follows is complete enough for the examples used in this thesis.

The processing of Notation3 files is modified by some *declarations*. In particular, @prefix prefix: <IRI> . is used to declare prefixes for Qnames. @keywords is used to change the parsing of keywords. These declarations should be at the top of the file. Anywhere else, and their behavior is not well specified.

Notation3 has *keywords*. We have already seen two of these. Then entire list of keywords currently is: keywords, is, of, has, prefix, base, a, forAll, forSome. Of these, a, has, is and of can appear just like that. The others must appear preceded by a @.

An IRI reference is put into <>. Thus, <http://www.w3.org/2000/10/swap/log#semantics> is an IRI reference RDF term.

A Qname simply appears as prefix:localname, like log:semantics. If preceded by the statement

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

then log:semantics is the same as <http://www.w3.org/2000/10/swap/log#semantics>, with the two strings concatenated. There is the empty prefix, which defaults to the

current document. Thus, : is valid, as is :a.

By using `@keywords keyword1, keyword2, ... .`, one can define which keywords must be preceded by `@`. By doing so, qnames with an empty prefix whose localname is not a keyword can appear without the :. Thus, after `@keywords a .`, `:a` must be written that way, but `:b` can be written as `b`. This can allow for much more readable notation3 files, while allowing for new keywords to be introduced in the future.

A string can be in double quotes `"This is a string"`, or triple double quotes

```
"""This is a
multiline string
"""
```

. A literal can be typed by being a number that is not in quotes, which is automatically typed, or a string followed by `^^dt`, where dt is a qname or IRI reference to the type. A language can be attached to a literal by appending `@langtag`, where langtag is the short name for the language, like "en". It is an error to include both a language tag and a type.

An RDF statement is a subject, predicate and object. In Notation3, one writes this as `subject predicate object .` . Multiple statements with the same subject can be combined using a `;`, while statements that differ only in the object can be combined using `,`. Thus, the following are the same.

```
:foo a :Thing1,
       :Thing2;
   :says "Hi" .


:foo a :Thing1 .
:foo a :Thing2 .
:foo :says "Hi" .
```

The keyword a stands for `<http://www.w3.org/1999/02/22-rdf-syntax-ns#type>`. Similarly, `=>` stands for `<http://www.w3.org/2000/10/swap/log#implies`, and `<=`

stands for is <http://www.w3.org/2000/10/swap/log#implies of. The keyword = stands for <http://www.w3.org/2002/07/owl#sameAs>.

Anonymous nodes, or blank nodes, have several representations. The first is uniquely named, using the special _ prefix in qnames. These will look like _:a16. The second is using [] notation. Anything within the brackets has the anonymous node as its subject. The third is using variable notation, declaring before any use of a IRI @forSome <IRI1>, foo:IRI2 . .

Another way yet of expressing blank nodes is using the path syntax. a!b is the same as [is b of a] . Also, a^b is the same as [b a] .

Notation3 has a shorthand for RDF linked lists. A list can be written as (elt1 elt2 ... eltn), as a shorthand for

```
_:L1 rdf:first elt1;
  rdf:rest _:L2 .
_:L2 rdf:first elt2;
  rdf:rest _:L3 .
...
_:LN rdf:first eltn;
  rdf:rest rdf:nil
```

A Notation3 graph can be quoted as a Notation3 node in a triple by putting it between { and }.

THIS PAGE INTENTIONALLY LEFT BLANK

Bibliography

[1] A. Kapadia and G. Sampemane and R. H. Campbell. Know why your access was denied: regulating feedback for usable security. In *11th ACM conference on Computer and Communications Security*, pages 52–61, 2004.

[2] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and interpretation of computer programs.* MIT Press, Cambridge, MA, USA, 1985.

[3] David Beckett. Turtle - Terse RDF Triple Language. `http://www.dajobe.org/2004/01/turtle/`.

[4] Tim Berners-Lee. Notation 3 (N3) A readable RDF Syntax. http://www.w3.org/DesignIssues/Notation3.html, 1998.

[5] Tim Berners-Lee. Cant.py canonicalize n-triples, 2003.

[6] Tim Berners-Lee. N3 BNF. `http://www.w3.org/2000/10/swap/grammar/n3-report.html`, 2006.

[7] Tim Berners-Lee, Dan Connolly, Lalana Kagal, Yosi Scharf, and Jim Hendler. N3logic: A logical framework for the world wide web. *TPLP*, 8(3):249–269, 2008.

[8] Tim Berners-Lee, Roy Fielding, and Larry Masinter. Uniform Resource Identifier (URI). `http://www.ietf.org/rfc/rfc3986.txt`, January 2005.

[9] Mark Birbeck and Shane McCarron. " curie syntax 1.0: A syntax for expressing compact uris". http://www.w3.org/TR/curie/, April 2008.

[10] T. Bray, D. Hollander, and A. Layman. "namespaces in xml w3c recommendation". http://www.w3.org/TR/1999/REC-xml-names-19990114, January 1999.

[11] Jeen Broekstra and Arjohn Kampman. Inferencing and Truth Maintenance in RDF Schema: exploring a naive practical approach. In *In Workshop on Practical and Scalable Semantic Systems (PSSS*, 2003.

[12] H. Chalupsky and T. Russ. Whynot: Debugging failed queries in large knowledge bases. In *Fourteenth Innovative Applications of Artificial Intelligence Conference (IAAI-02)*, pages 870–877, 2002.

[13] Weidong Chen and David S. Warren. Tabled evaluation with delaying for general logic programs. *J. ACM*, 43(1):20–74, 1996.

[14] Johan de Kleer, Jon Doyle, Jr. Guy L. Steele, and Gerald Jay Sussman. AMORD Explicit Control of Reasoning. *SIGPLAN Not.*, 12(8):116–125, 1977.

[15] python-dlp. http://code.google.com/p/python-dlp/.

[16] Robert B. Doorenbos. *Production matching for large learning systems.* PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 1995.

[17] Jon Doyle. A truth maintenance system. *Artificial Intelligence*, 12(3):231–272, November 1979.

[18] Kenneth D. Forbus and Johan de Kleer. *Building problem solvers.* MIT Press, Cambridge, MA, USA, 1993.

[19] Charles L. Forgy. RETE: A fast algorithm for the many pattern/many object match problem. *Artificial Intelligence*, 19:17–37, September 1982.

[20] Fuxisemantics. http://code.google.com/p/python-dlp/wiki/FuXiSemantics.

[21] Michael Gelfond and Vladimir Lifschitz. The stable model semantics for logic programming. In Robert A. Kowalski and Kenneth Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 1070–1080, Cambridge, Massachusetts, 1988. The MIT Press.

[22] MIT Distributed Information Group. Transparent Accountable Datamining Initiative. http://dig.csail.mit.edu/TAMI/.

[23] Chris Hanson and Lalana Kagal. AIR Policy Language. http://dig.csail.mit.edu/TAMI/2007/amord/air-specs.html, 2007.

[24] Brian Hayes. Unwed numbers. *American Scientist*, 94:12–15, 2006.

[25] Patrick Hayes. RDF Semantics. W3C Recommendation. http://www.w3.org/TR/rdf-mt/, February 2004.

[26] Ian Horrocks, Oliver Kutz, and Ulrike Sattler. The Even More Irresistible SROIQ. In *Proc. of the 10th Int. Conf. on Principles of Knowledge Representation and Reasoning (KR2006)*, pages 57–67. 10th International Conference on Principles of Knowledge Representation and Reasoning, AAAI Press, June 2006.

[27] Ian Horrocks, Peter F. Patel-Schneider, Harold Boley, Said Tabet, Benjamin Grosof, and Mike Dean. SWRL: Semantic Web Rule Language Combining OWL and RuleML. http://www.daml.org/rules/proposal/, 2004.

[28] Keith L. Clark. Negation as Failure. *Logic and Data Bases*, 1978.

[29] Brian Kelley. Graph canonicalization. *j-DDJ*, 28(5):66–69, may 2003.

[30] Robert A. Kowalski. Predicate logic as programming language. In *IFIP Congress*, pages 569–574, 1974.

[31] Deborah L. Mcguinness and Paulo Pinheiro. Explaining Answers from the Semantic Web: the Inference Web Approach. *Web Semantics: Science, Services and Agents on the World Wide Web*, 1(4):397–413, October 2004.

[32] Daniel P. Miranker. TREAT: A Better Match Algorithm for AI Production Systems; Long Version. Technical report, University of Texas at Austin, Austin, TX, USA, 1987.

[33] Bijan Parsia, Yarden Katz, and Kendall Clark. Pychinko: Rete-based RDF friendly rule engine. http://www.mindswap.org/~katz/pychinko/, January 2005.

[34] Optimizing SPARQL-DL. http://clarkparsia.com/weblog/2007/11/02/optimizing-sparql-dl/.

[35] Eric Prud'hommeaux and Andy Seaborne. SPARQL Query Language for RDF. W3C Recommendation. http://www.w3.org/TR/rdf-sparql-query/, January 2008.

[36] Jos De Roo. Euler proof mechanism. http://www.agfa.com/w3c/euler/, 2005.

[37] Bertrand Russell. Letter to frege (1902), 1967.

[38] Evren Sirin, Bijan Parsia, Bernardo Cuenca Grau, Aditya Kalyanpur, and Yarden Katz. Pellet: A practical OWL-DL reasoner. *Web Semant.*, 5(2):51–53, 2007.

[39] Gerald Sussman and Terry Winograd. Micro-Planner Reference Manual. Technical report, Massachusetts Institute of Technology, Cambridge, MA, USA, 1970.

[40] Allen van Gelder, Kenneth Ross, and John S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

[41] Guido van Rossum. The Official Python Programming Language Website. http://www.python.org/.

[42] W3C. W3C Semantic Web Best Practices and Deployment Working Group. http://www.w3.org/2001/sw/BestPractices/.

[43] W3C. RDF/XML Syntax Specification (Revised). http://www.w3.org/TR/rdf-syntax-grammar/, 2004.