10-19-2015

An extensible natural-language query interface to the DBpedia Triple-store

Wale Agboola
University of Windsor

AN EXTENSIBLE NATURAL-LANGUAGE QUERY INTERFACE TO THE DBPEDIA TRIPLE-STORE

by

Wale Agboola

A Thesis
Submitted to the Faculty of Graduate Studies
through the School of Computer Science
in Partial Fulfilment of the Requirements for
the Degree of Master of Science
at the University of Windsor

Windsor, Ontario, Canada

2015

AN EXTENSIBLE NATURAL-LANGUAGE QUERY INTERFACE TO THE DBPEDIA TRIPLE-STORE

by

Wale Agboola

APPROVED BY:

Dr. Richard J. Caron, Outside Department Reader
Mathematics and Statistics Department

Dr. Ziad Kobti, Inside Department Reader
School of Computer Science

Dr. Scott Goodwin, Inside Department Reader
School of Computer Science

Dr. Richard A. Frost, Advisor
School of Computer Science

August 27, 2015

DECLARATION OF CO-AUTHORSHIP / PREVIOUS PUBLICATION

I hereby declare that this thesis incorporates the outcome of a joint research undertaken in collaboration with Jonathon Donais, Eric Mathews, and Rob Stewart under the supervision of Dr. Richard A. Frost.

The collaboration is covered in Chapter 3.2 of the thesis. In all cases, the key ideas, primary contributions, experimental designs, data analysis and interpretation, were performed by the author, and the contribution of co-authors was primarily through the provision of Dr. Richard A. Frost.

I am aware of the University of Windsor Senate Policy on Authorship and I certify that I have properly acknowledged the contribution of other researchers to my thesis, and have obtained written permission from each of the co-author(s) to include the above material(s) in my thesis.

I certify that, with the above qualification, this thesis, and the research to which it refers, is the product of my own work. This thesis incorporates two original papers that have been previously published / submitted for publication in peer reviewed journals, as follows:

Thesis Chapter	**Publication title / full citation**	**Publication status**
2	Frost, R. A., Agboola, W., Matthews, E., and Donais, J. (2014a). An event-driven approach for querying graph-structured data using natural language. In *Querying Graph Structured Data (GraphQ)), Organization=EDBT/ICDT 2014 Joint Conference, pages=192–199*	published
2	Frost, R. A., Donais, J., Matthews, E., Agboola, W., and Stewart, R. (2014b). A demonstration of a natural language query interface to an event-based semantic web triplestore. In *The Semantic Web: ESWC 2014 Satellite Events*, pages 343–348. Springer International Publishing	published

I certify that I have obtained a written permission from the copyright owner(s) to include the above published material(s) in my thesis. I certify that the above material describes work completed during my registration as graduate student at the University of Windsor.

I declare that, to the best of my knowledge, my thesis does not infringe upon anyone's copyright nor violate any proprietary rights and that any ideas, techniques, quotations, or any other material from the work of other people included in my thesis, published or otherwise, are fully acknowledged in accordance with the standard referencing practices. Furthermore, to the extent that I have included copyrighted material that surpasses the bounds of fair dealing within the meaning of the Canada Copyright Act, I certify that I have obtained a written permission from the copyright owner(s) to include such material(s) in my thesis. I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee and the Graduate Studies office, and that this thesis has not been submitted for a higher degree to any other University or Institution.

ABSTRACT

DBpedia is a triple-based binary-relational database which contains 3 billion, and counting, facts derived from Wikipedia. Ideally, individuals should be able to access semantic web triple-store data through natural-language queries. Several attempts have been made to create natural-language (NL) query interfaces to DBpedia. However, no one has yet built a wide-coverage natural-language query processor for DBpedia. DBpedia does not currently encode contextual data representing the time, location or other properties of binary-relationships. This means that NL queries cannot contain prepositional phrases such as the phrase "in 2004" in the query: "what film was directed by Clint Eastwood in 2004". Existing NL query interfaces to DBpedia cannot handle prepositional phrases; they are also unable to be extended to do so when used with triple-stores other than DBpedia, which can accommodate contextual data. In this thesis, we investigate an alternative approach to querying DBpedia in which NL queries are treated as expressions of the lambda calculus which are evaluated directly with respect to the triple-store using a compositional and extensible denotational semantics of English.

ACKNOWLEDGEMENT

This project would not have been possible without the support of many people. Many thanks to my advisor, Dr. Richard A. Frost, who read my numerous revisions and guided my thesis study, and provided me with financial support from the Natural Science and Engineering Research Council of Canada (NSERC).

I would also like to thank Rob Stewart from Heriot-Watt University in Edinburgh, Scotland.

I would also like to thank Dr. Richard J. Caron, Dr. Ziad Kobti and Dr. Scott Goodwin for reading my thesis report and for their valuable comments. And finally, I would like to thank my family and friends who endured this long process with me, always offering support.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF SOURCE CODES

LIST OF APPENDICES

NOMENCLATURE

DBpedia	DataBase-pedia
DEV-NLQ	Direct Evaluation of Natural Language Queries
MS	Richard Montague's Semantics
NL	Natural Language
NLQI	Natural Language Query Interface
NLQIs	Natural Language Query Interfaces
RDF	Resource Description Framework
SPARQL	SPARQL Protocol and RDF Query Language
YAGO	Yet Another Great Ontology

1. INTRODUCTION

1.1 What is DBpedia?

DBpedia ("database-pedia") is a project aiming to extract structured content from the on-line text information created as part of the Wikipedia project. This structured information is available on the World Wide Web. DBpedia allows users to query relationships and properties associated with Wikipedia resources, including links to other related datasets. The current DBpedia (version 2014) data set describes 4.58 million entities, including 1,445,000 persons, 735,000 places, 123,000 music albums, 87,000 films, 19,000 video games, 241,000 organizations, 251,000 species and 6,000 diseases. The DBpedia project uses the Resource Description Framework (RDF) to represent the extracted information and consists of 3 billion RDF triples, 580 million extracted from the English edition of Wikipedia and 2.46 billion from other language editions.

The datasets are stored in a set of RDF triples. An RDF Triple is a statement which relates one object to a subject using a predicate in the form, <Subject> <Predicate> <Object>. For example, facts about Clint Eastwood may include the following triples in DBpedia the dataset:

```
<dbpedia:Clint_Eastwood> <dbpedia-owl:birthplace> <dbpedia:San_Francisco>.
<dbpedia:Gran_Torino> <dbprop:director> <dbpedia:Clint_Eastwood>.
<dbpedia:Heartbreak_Ridge> <dbprop:director> <dbpedia:Clint_Eastwood>.
<dbpedia:J.Edgar> <dbprop:director> <dbpedia:Clint_Eastwood>.
<dbpedia:Unforgiven> <dbprop:director> <dbpedia:Clint_Eastwood>.
<dbpedia:Million_Dollar_Baby> <dbprop:director> <dbpedia:Clint_Eastwood>.
```

Listing 1: Sample triples: facts about Clint Eastwood

In the set of triples, *dbpedia*, as in <dbpedia:Clint_Eastwood>, represents a resource name prefix in DBpedia; *dbpedia-owl*, as in <dbpedia-owl:birthplace>, represents an ontology prefix used as a bridge between to resources; and *dbprop*, as in <dbprop:director>, represents a property prefix used as a bridge between to resources.

Based on the set of triples, it is noted that Clint Eastwood was born in San Francisco. It is also noted that Clint Eastwood directed the films Gran Torino, Heartbreak Ridge, J. Edgar, Unforgiven and Million Dollar Baby.

Each identifier in each triple is a uniform resource identifier (URI), which is a string of characters used to identify the name of a resource.

1.2 Overview of natural-language query interfaces to DBpedia

Many researchers in the Semantic Web field have designed Natural Language Interfaces to DBpedia. Tablan et al. [17] claims to have designed an NLI system, QuestIO, that is able to accept a wide range of syntactically ill-formed queries or short fragments and convert them into formal queries that can be executed against a knowledge store. Lehmann et al. [13] claims to have designed a system, DEQA, that improves existing search functionality by combining web extraction, data integration and enrichment as well as question answering. Unger et al. [18] claim to have a template based NLI system than can generate SPARQL templates to capture the semantic structure of the natural language input provided by the user. Lopez et al. [14] designed an NLI system, PowerAqua, that can answer user's requests extending beyond the coverage of single datasets. Damljanovic et al. [5] designed an NLI system, FREyA, that is portable and was tested with both MusicBrainz and DBpedia datasets, and claimed to produces better results than PowerAqua [14]. Wan et al. [19] claim to have designed an NLI system that allows querying of semantic information from multiple sources through a unified user-friendly interface and automatic data integration to improve the coverage and accuracy of users information query.

These Natural Language Interfaces use SPARQL (SPARQL Protocol and RDF Query Language), a semantic query language for databases, that is able to retrieve and manipulate data stored in the Resource Description Framework (RDF) format. However, the use of SPARQL, as well as DBpedia triple-stores, limits the expressive power of Natural Language Interfaces.

1.3 The problem

Firstly, most existing Natural-Language Query Interfaces (NLQIs) such as FREyA [5], PowerAqua [14] and QuestIO [17] have limited expressive power and cannot handle queries involving chained complex prepositional phrases and arbitrarily-nested quantification. For example, it appears to be impossible for these Natural Language Query Interfaces to translate the query "who stole a car in Manhattan in 1918 or 1920?" to SPARQL because there is no contextual properties of relationships between entities, namely time and location.

```
<"Al_Capone"> <"steal"> <"car_1">.
<"car_1"> <"steal_Date"> <"1918">.
<"car_1"> <"steal_Location"> <"Manhattan">.
<"car_1"> <"steal_Location"> <"London"> ...
```

In other words, there might be a triple-store data representation in a Semantic Web triple-store that "*Al Capone stole car 1*", but the triple is not a direct representation that "*Al Capone stole car 1 in Manhattan in 1918*". Furthermore, there might be a triple-store data representation that "*car 1 was stolen in 1918*", but the two triples do not represent the fact that "*car 1 was stolen by Al Capone in Manhattan*" because there is nothing to link the triples together.

Consequentially, no one has yet been able to successfully translate complex natural language queries with prepositional phrases to SPARQL.

1.4 Emerging, more expressive triple-stores

1.4.1 DEV-NLQ

Frost et al. [8]'s Direct Evaluation of Natural Language Queries (DEV-NLQ) defines and uses event-based triple-stores, treating bracketed English queries as expressions of the lambda calculus which can be evaluated directly with respect to the triple-store. For example the fact that *Al Capone stole car_1* is represented as follows:

```
<EV 1001> <REL "type"> <TYPE "steal_ev"> .
<EV 1001> <REL "subject"> <ENT "Al Capone"> .
<EV 1001> <REL "object"> <ENT "car_1"> .
```

The fact that Al Capone stole car_1 in 1918 can now be represented by adding the following triple:

```
<EV 1001> <REL "year"> <ENTNUM 1918> .
```

'Al Capone stole car_1 in 1918' is now represented by a single event: <EV 1001>.

Furthermore, Frost et al. [8] noted that no method, other than theirs, can accommodate NL queries, such as the "Who stole a car in 1918 in Manhattan?", which contain chained complex prepositional phrases.

Quantifiers are words or phrases which indicate the number or amount being referred to. Such quantifiers include "a", "one", "two", "every", and "no".

1.4.2 YAGO2

Hoffart et al. [11] noted that a major drawback for querying DBpedia with SPARQL is that even a small query on a particular object required convoluted joins. Hoffart et al.'s YAGO2 [11] is a Semantic Web Knowledge base with a focus on temporal and spatial knowledge. It is automatically built from Wikipedia, GeoNames, and Word-Net, and contains nearly 10 million entities and events, as well as 80 million facts representing general world knowledge. A new model is introduced in the form of SPOTL(X) View; an extended 5-tuple containing SPO triples augmented by Time, Location and eXplanation.

An example of YAGO2's dataset containing the extended 5-tuples is shown in table 1.1.

Tab. 1.1: Example of YAGO2 data-store

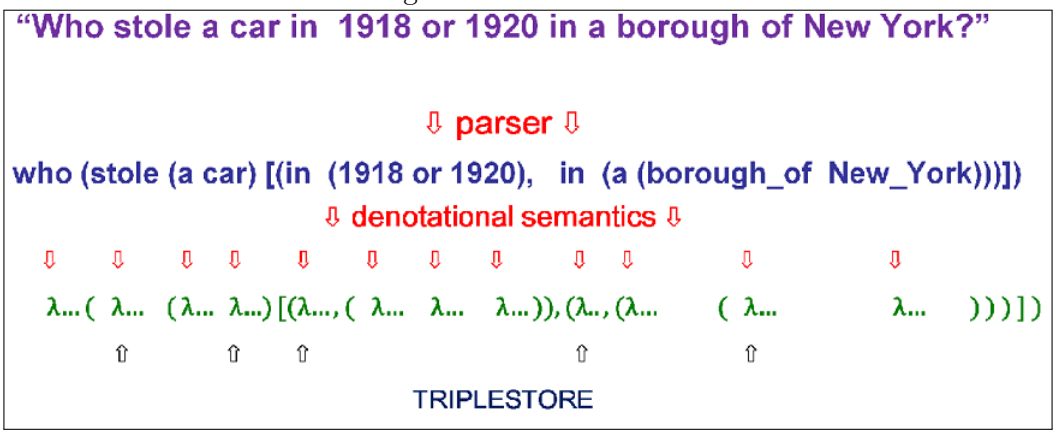
Id	Subject	Property	Object	Time	Location	Keywords
id1	Al Capone	steal	car_1	1918	Manhattan	
id2	Al Capone	born	USA	1899	New York	
id3	Gran Torino	director	Clint Eastwood	2008	Michigan	Detroit, Grand Rapids, Grosse Pointe, Royal Oak
id4	Heartbreak Ridge	director	Clint Eastwood	1986	California	San Clemente, Agua Dulce, Santa Clarita, El Toro, San Diego, San Juan Capistrano
id5	J. Edgar	director	Clint Eastwood	2011	USA	District of Columbia, California, Virginia, USA
id6	Unforgiven	director	Clint Eastwood	1992	Alberta, Canada	Brooks, Longview, Drumheller, High River, Calgary
id7	Million Dollar Baby	director	Clint Eastwood	2004	USA	California, Nevada

As seen in table 1.1, YAGO2 extends triples to 5-tuples which allow context to be added to the facts. Furthermore, YAGO2 use a technique called reification to represent contextual data. However, reification further complicates the translation to SPARQL. Therefore, an alternative approach is to treat the Natural Language queries as expressions of the Lambda Calculus and evaluate them directly with respect to the triple-stores.

1.5 A new approach

The new approach involves a natural language query interface to DBpedia triple-store that treat words and phrases in the query as expressions of the lambda calculus which are applied to each other according to the syntactic structure of the query.

Fig. 1.1: The basic idea



Where λ are functional denotations of words based on an efficient version of Montague Semantics [6]. The $\lambda \ldots \ (\ \lambda \ldots \ (\ \lambda \ldots \ \lambda \ldots) \ [(\lambda \ldots \ , \ (\ \lambda \ldots .$ above is an expression of the lambda calculus. The lambda expressions are implemented and evaluated directly in the Haskell programming language. Some functions (indicated by $\Uparrow$ in the above) are defined in terms of triple-store retrieval operations.

This approach is based on Montague's Semantics (MS) of natural language in which words denote functions, and the meaning of a phrase is computed by applying those functions to each other according to the syntactic structure of the phrase. This approach can be modified to work with more expressive triple-stores such as YAGO2 [11] and emerging event-based triple-stores [7], which can represent contextual data about facts, such as the time and location of an event, which DBpedia cannot.

In other words, when a simple query is asked, that query is input to a parser [9]. The parser then generates the denotation and meaning of words, then the functional expression denoted by the query is evaluated with respect to the DBpedia triple-store.

1.6 The thesis statement

It is possible to construct an extensible, usable, wide coverage natural language query interface to DBpedia triple-store by treating words and phrases in the query

as expressions of the lambda calculus which are applied to each other according to the syntactic structure of the query.

By *usable*, it is possible for answers to be returned in less than 100 seconds.

By *wide coverage*, the queries can include common nouns, proper nouns, adjectives, "and", "or", "then", intransitive and transitive verbs, and nested quantification with quantifiers: "a", "one", "two", "every", and "no".

By *extensible*, the approach can be extended for use with more powerful triple-stores, such as YAGO2 and event-based triple-stores.

1.7 Importance of thesis statement

Firstly, most NLQIs convert a Natural Language (NL) query to a SPARQL [15] query which is then evaluated with respect to one or more triple-stores. The new approach uses a method of retrieving triples and manipulating those triples with respect to each other. Furthermore, theories for building extensible query processors can be implemented. When more expressive triple-stores become available, the query processor can be extended to accommodate prepositional phrases, as in the query "what did Al Capone steal in 1918?". Secondly, most semantic web triple-stores cannot represent "contextual" properties of relationships between entities such as time, location, and context without using some form of "reification" – the reification significantly complicates the translation to SPARQL, and lastly, no-one has yet been able to translate a NL query such as "who stole a car in Manhattan in 1918 or 1920" to SPARQL.

1.8 Non-triviality of thesis statement

Most NLQIs translate NL queries to SPARQL and answers are presented to the users. However, these NLQIs cannot answer complex questions that deal with complex prepositional phrases. We appear to be the first to treat NL queries as expressions of the lambda calculus and evaluate them directly with respect to DBpedia.

The functions which are the meaning of words are defined in the Haskell programming language. This approach to Natural Language interfaces has not

been implemented by any other research group, therefore, we are the first to use the Haskell HSPARQL [20] package in a triple-store query interface.

1.9 Proof of thesis statement

To prove the thesis, a new modular and extensible approach for building Natural Language Query Interfaces to triple-stores will be implemented:

- that treats the (bracketed) queries as expressions of the lambda calculus.
- evaluates the queries directly with respect to the DBpedia triple-store.
- Time query evaluation followed by optimization.
- Show how the approach could be extended to accommodate prepositional phrases if the triple-store represents contextual data.

1.10 Structure of the thesis report

The rest of the thesis paper is structured as follows: Chapter 2 contains a summary of related work done by other research groups, as well as related work by members of the University of Windsor's Speechweb research group. Chapter 3 fully describes the Semantics. Chapter 4 describes the time evaluation of queries. Chapter 5 analyses and shows the time complexities. Chapter 6 contains recommendations for reducing the computational cost of the semantics described in this paper. Chapter 7 explains how the approach described in this thesis can be extended to accommodate prepositional phrases. Lastly, chapter 8 concludes the thesis and discusses future work.

2. RELATED WORK

2.1 Related work by other NLQI researchers

Papers which are closely related to this thesis include [5, 12, 13, 14, 17, 18, 19]. These papers are all concerned with building and implementing Natural Language Query Interfaces to structured information on the Semantic Web. These papers describe systems that are able to accept a wide range of queries or short fragments and convert them into formal queries that can be executed against a knowledge store. Consequentially, the methods proposed in these papers have proven to be essential for further research into Question Answering Systems for the Semantic Web. These papers also concern architectures that are able to allow queries from users to be expressed in natural language, aggregate and rank answers drawn from relevant distributed resources on the Semantic Web, such as MusicBrainz [16] and DBpedia. Furthermore, these architectures extend far beyond single domains across different sources and domains.

2.2 Event-based triples and DEV-NLQ

Frost et al., including the author of this Master's thesis, published a paper [7] concerning the use of events rather than entities as the subject of triples and treating (bracketed) Natural Language queries as expressions of the lambda calculus. Furthermore, Natural Language queries are translated to functional expressions, rather than an intermediate language such as SPARQL. Another paper was published by Frost's research group [8]. This paper focuses on the concept that Natural Language semantic web queries can be evaluated directly with respect to an event-based triple-store using only basic triple retrieval operations, which then facilitates the accommodation of complex Natural Language constructs. DEV-NLQ, Direct Evaluation of Natural Language Queries, uses event-based triple-stores, treating bracketed English queries as expressions of the lambda calculus which can be evaluated directly with respect to the triple-store.

However, this thesis uses entity-based triple-stores from DBpedia, treating bracketed English queries as expressions of the lambda calculus which can be

evaluated directly with respect to DBpedia's triple-stores.

3. THE SEMANTICS

3.1 The new idea

The idea presented in the semantics treats bracketed English queries as expressions of the lambda calculus which are evaluated directly with respect to triple-stores. The semantics accommodate proper and common nouns, adjectives, intransitive and transitive verbs, negation, and chained complex prepositional phrases containing arbitrarily-nested quantifiers. Furthermore, each word in English denotes a function.

Firstly, words that correspond to DBpedia prefixes that are part of the RDF identifiers are defined.

```
namespace_prop = "http://dbpedia.org/property/"
namespace_res = "http://dbpedia.org/resource/"
namespace_ont = "http://dbpedia.org/ontology/"
namespace_ctgry = "http://dbpedia.org/resource/Category:"
namespace_umbel = "http://umbel.org/umbel/rc/"
namespace_yago = "http://dbpedia.org/class/yago/"
type0 = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
subject = "http://purl.org/dc/terms/subject"
prop fragment = namespace_prop ++ fragment
res fragment = namespace_res ++ fragment
ont fragment = namespace_ont ++ fragment
ctgry fragment = namespace_ctgry ++ fragment
umbel fragment = namespace_umbel ++ fragment
yago fragment = namespace_yago ++ fragment
```

Listing 2: DBpedia prefixes denotations

Common nouns such as actors, producers, and directors are defined. Each of these functions as well as others are defined using Haskell programming language. Simple *getts* function is used to retrieve triples from DBpedia using the HSPARQL package. HSPARQL is a Haskell module that allows the Haskell program to

execute SPARQL queries against a remote triple-store through a remote SPARQL endpoint. However, we only use HSPARQL to issue basic triple retrieval requests, not complex SPARQL queries. We are thereby able to take advantage of the efficient retrieval capabilities of remote SPARQL endpoints without having to translate the NL query to SPARQL.

```
--films
film = getts_1 ("?", type0, ont "Film")
--actors
actor = getts_1 ("?", subject, ctgry "American_film_actors")
--producers
producer = getts_1 ("?", subject, ctgry "American_film_producers")
--directors
director = getts_1 ("?", subject, ctgry "American_film_directors")
```

Listing 3: Common nouns denotations

The queries are not converted to SPARQL. In order to execute these SPARQL functions, the HSPARQL package [20], is used to connect to a specified SPARQL endpoint and retrieve triples. These triples can then be manipulated directly without the complicated task of translating to SPARQL.

Using Montague semantics [6], the meaning of a sentence in English is composed from the meanings of its component words and phrases. The Montague semantics is extended for more complex Natural Language queries. Proper nouns are defined in terms of entities. For example, the proper noun *Clint Eastwood* denotes the entity in DBpedia associated with "Clint_Eastwood", which is `http://dbpedia.org/resource/Clint_Eastwood`.

The phrase "Clint Eastwood directs" is interpreted as shown below, where $xy = z$ indicates that the result of xy is equal to the evaluation of z. Proper nouns are defined to return a boolean answer. If the entity (e.g. Clint Eastwood), is in the list of entities (e.g. Directors), it returns True, and False, otherwise.

```
--proper nouns
--every noun is defined
brad_pitt setofents = checkMember (res "Brad_Pitt") setofents
angelina_jolie setofents = checkMember (res "Angelina_Jolie") setofents
chuck_lorre setofents = checkMember (res "Chuck_Lorre") setofents
clint_eastwood setofents = checkMember (res "Clint_Eastwood") setofents
```

Listing 4: Proper noun denotations

Furthermore, there are functions that are defined for the quantifiers "every", "a", "one", "two", etc.

```
a nph vbph = length (interset nph vbph) /= 0
every nph vbph = subset nph vbph
one nph vbph = length(interset  nph vbph) == 1
two nph vbph = length(interset  nph vbph) == 2
which nph vbph = interset nph vbph
how_many nph vbph = length (interset nph vbph)
```

Listing 5: Quantifiers

```
produced tmph = [subject | (subject, objects) <-
                                reverse_image2 "producer",
                                tmph objects ]
```

Listing 6: Transitive verb denotation: *produced*

```
directed tmph  = [ subject | (subject, objects) <-
                                reverse_image2 "director",
                                tmph objects ]
```

Listing 7: Transitive verb denotation: *directed*

3.1.1 *Interfacing the NLQ processor to DBpedia SPARQL endpoint using HSPARQL*

HSPARQL [20] is a Haskell package that allows Haskell programs to interface remotely with the Semantic Web triple-stores. HSPARQL supports SELECT, CONSTRUCT, ASK and UPDATE queries. However, for this thesis and the query interface, only simple triple retrieval SPARQL queries are used.

```
SELECT ?first WHERE {?first, <given_second>, <given-third>} .
SELECT ?third WHERE {<given_first>, <given_second>, ?third} .
SELECT ?first ?third WHERE {?first, <given_second>, ?third} .
```

Triple-stores are accessed through the Virtuoso "SPARQL endpoint". In order to access these triple-stores, the SPARQL endpoint must be made available over the internet. For example, Frost et al.'s event-based triple-stores [7] can be accessed at `http://speechweb2.cs.uwindsor.ca/sparql`, therefore its SPARQL endpoint is `http://speechweb2.cs.uwindsor.ca/sparql`. In the case of DBpedia, its SPARQL endpoint is located at `http://dbpedia.org/sparql` or `http://live.dbpedia.org/sparql`.

```
film = getts_1 ("?", type0, ont "Film") ---List of films
-- getts_1 (x, y, z) returns a set x, given y and z.
```

Listing 8: A simple haskell function interface with HSPARQL

Furthermore, the Haskell package HSPARQL [20] is used to interface the query processor to the external SPARQL endpoint containing the DBpedia data. Data retrieval functions, such as the *getts_1* above, are defined in terms of HSPARQL functions in a module called Getts_v7. (see Appendix A).

There are two modules that must be imported to use the HSPARQL package in Haskell.

```
import Database.HSparql.Connection
import Database.HSparql.QueryGenerator
```

The first package, Connection, creates the connection to the SPARQL endpoint, while QueryGenerator package, the query generator for SPARQL, is used when connecting to a remote endpoint.

All strings, such as "Al_Capone" or "Clint_Eastwood" in the definitions are modified by a function *res* or *prop* to include a full resource or property URI prefix of the String. URI (Uniform Resource Identifier) is a world-wide unique identifier used in the semantic web to resolve ambiguities in entity naming. For example, the URI for "Clint_Eastwood" is `http://dbpedia.org/resource/Clint_Eastwood`.

3.1.2 Example denotations of words

Common Nouns

Common nouns are defined as functions that denote a class of objects or a concept as opposed to a particular individual. For example, the common noun *film* denotes a function that returns all subjects listed as "English-languageFilms" in DBpedia.

```
--films
film = getts_1 ("?", type0, yago "English-languageFilms")
```

Listing 9: Common noun: *film*

The prefix "yago" represents a function to recognize DBpedia prefixes that are part of the RDF identifiers. The url "`http://dbpedia.org/class/yago/English-languageFilms`" is the RDF identifier for "English-languageFilms", and *yago "English-languageFilms"* used to identify "English-languageFilms" as an RDF resource.

Another example is the common noun *actor*. The *actor* function returns all subjects that are listed as actors in DBpedia.

```
--actors
actor = getts_1 ("?", type0, yago "AmericanFilmActors")
```

Listing 10: Common noun: *actor*

Proper nouns

Proper nouns are related to a person, place, thing, or idea. In this thesis paper, proper nouns include only specific names. Examples of proper nouns are *Million Dollar Baby*, *Django Unchained*, and *Eat Pray Love*. Each of these proper nouns is a *film*. Note that *film* is a common noun. Using Montague Semantics, proper nouns do not denote entities directly. Rather, proper nouns denote functions that take a set of entities as an argument and which return True if a particular entity is a member of that set. For example, the proper noun "Million_Dollar_Baby" is denoted by the function *million_dollar_baby* where *"million_dollar_baby"* represents the entity associated with the URI name for "Million Dollar Baby". Therefore *million_dollar_baby setofents* returns true if *"million_dollar_baby"* is in the list *setofents*.

```
--proper nouns
brad_pitt setofents
        = checkMember (res "Brad_Pitt") setofents
quentin_tarantino setofents
        = checkMember (res "Quentin_Tarantino") setofents
django_unchained setofents
        = checkMember (res "Django_Unchained") setofents
eat_pray_love setofents
        = checkMember (res "Eat_Pray_Love") setofents
clint_eastwood setofents
        = checkMember (res "Clint_Eastwood") setofents
million_dollar_baby setofents
        = checkMember (res "Million_Dollar_Baby") setofents
pulp_fiction setofents
        = checkMember (res "Pulp_Fiction") setofents
```

Listing 11: Proper nouns

Conjunctions: "and" and "or"

A conjunction is a part of speech that connects words, sentences, phrases, or clauses. Conjunctions are defined as functions that take words, phrases or term phrases, and returns the boolean answer based on the contents of the argument passed and context of the conjunction.

```
--returns a set that is common in both input sets s and t
nounand s t = interset s t
--returns a union set of both input sets s and t
nounor s t = makeset(s ++ t)
--Finds a common set given 2 sets
--Complexity => O(n+m)
interset s t = List.intersect s t
```

Listing 12: Conjunction words

Determiners

Determiners are used before nouns in a term phrase. That is to say, a determiner will be followed by a noun. For example, in the phrase *direct a film*, *"a"* is the determiner.

Furthermore, determiners are defined as functions that take term phrases and returns a boolean based on the definition of that particular determiner.

```
a nph vbph = length (interset nph vbph) /= 0
--returns true if nph and vbph have at least 1 item in common
every nph vbph = subset nph vbph
--returns true if nph set is contained in vbph
one nph vbph = length(interset  nph vbph) == 1
--returns true if nph and vbph have exactly 1 item in common
two nph vbph = length(interset  nph vbph) == 2
--returns true if nph and vbph have exactly 2 items in common
```

Listing 13: Determiners

Transitive verbs

Transitive verbs are explicitly defined such that they have a term phrase to receive that action or verb. For example, the denotation of *directed* (as in "directed a film") is shown below, in which the *directed* predicate corresponds to the predicate URI representing "director" in DBpedia.

```
directed tmph = [ subject | (subject, objects) <-
                                    reverse_image2 "director",
                                    tmph objects]
```

Listing 14: Transitive verb: *directed*

Informally, the definition of *directed* uses relative set notation.

$[\ subject \mid b1 = (subject,\ objects) \in set1,\ c1]$ is read as the set of all subject such that $b1$ is a member of the set $set1$ and $c1$ is condition.

The function *reverse_image2* is defined such that it returns a new binary-relation. Furthermore, *reverse_image2 "director"* returns all (s)ubject, (o)bject pairs with the (p)redicate <dbprop:director> from DBpedia.

[(x1, y1), (x2, y2),...]

The pairs in the list are then reversed:

[(y1, x1), (y2, x2),...]

Reversing the pairs is only necessary if the pairs are not in the correct pair order. For example, the pairs for the "director" predicate returns a list:

[(film1, director1), (film2, director2),...], but the correct result required is ***[(director1, film1), (director2, film2),...]***.

The pairs are then processed by a "*collect*" function that creates a collection of images where each item in the list is a pair of directors and the list of films they directed. E.g = ***[(director1, [film1, film2, film3, ...]), ...]***.

For example, consider the following triples:

```
{...
<dbpedia:Gran_Torino> <dbprop:director> <dbpedia:Clint_Eastwood> .
<dbpedia:Heartbreak_Ridge> <dbprop:director> <dbpedia:Clint_Eastwood> .
<dbpedia:J.Edgar> <dbprop:director> <dbpedia:Clint_Eastwood> .
<dbpedia:Unforgiven> <dbprop:director> <dbpedia:Clint_Eastwood> .
<dbpedia:Million_Dollar_Baby> <dbprop:director> <dbpedia:Clint_Eastwood>.
...}
```

Then

```
reverse_image2 "director" = [..., (<dbpedia:Clint_Eastwood>,
        [<dbpedia:Gran_Torino>, <dbpedia:Heartbreak_Ridge>,
        <dbpedia:J.Edgar>, <dbpedia:Unforgiven>,
        <dbpedia:Million_Dollar_Baby>]), ... etc]
```

Note that the ***tmph*** in *directed tmph* denotes a term phrase. This term phrase may include a proper noun or a phrase within a quantifier. For example, the query can be "*directed million_dollar_baby*" or "*directed (a film)*".

The term phrase *tmph* is applied to the image associated with the subjects by a function, and returns the subject if the evaluation of *tmph* with the subject's set of objects is True. For example:

```
directed (a film)
                = [ subject | (subject, objects) <-
                                reverse_image2 "director",
                                a film objects ]
```

In the code snippet above, anything before "|" is the result set expected, "<-" is a generator, and the expressions after ',' are conditions. This means that each subject-objects pair is retrieved from the result of *reverse_image2 "director"*, and for each pair, the condition ***a film objects*** is checked. If the condition is True, then add the *subject* of the subject-objects pair to the result set.

The query "directed (a film)" takes about 80 seconds to complete. For every objects list in the subject-objects pairs, the 'a' function checks if there is at least a common element in both $film$ and $objects$. The 'a' function, as described on page 17, makes use of the intersect function in the List module [1]. Its complexity is $\mathcal{O}(n+m)$, where m is the number of elements in the film list and n is the number of elements in objects list.

```
directed million_dollar_baby
                = [ subject | (subject, objects) <-
                        reverse_image2 "director",
                        million_dollar_baby objects ]
```

The query "directed million_dollar_baby" takes about 7 seconds to complete. For every objects list in the subject-objects pairs, the "million_dollar_baby" function checks if the URI representing the entity 'Million Dollar Baby' is in the list $objects$. Its complexity is $\mathcal{O}(n)$, where n is the number of elements in objects list.

```
directed (two films)
                 = [ subject | (subject, objects) <-
                                reverse_image2 "director",
                                two films objects ]
```

The query "directed (two films)" takes about 80 seconds to complete. For every objects list in the subject-objects pairs, the 'a' function checks if there are exactly 2 common elements in both $film$ and $objects$. The 'two' function, as described in 17, makes use of the intersect function in the List module [1]. Its complexity is $\mathcal{O}(n+m)$, where m is the number of elements in the film list and n is the number of elements in objects list.

The denotation of transitive verbs makes the semantics computationally expensive: First, the pairs are retrieved from the triple-store. Second, the image function creates a subject-objects map from the pairs. Third, the term phrase is evaluated with the list of objects associated with a subject. Fourth, for all evaluations that are True, return the subject.

The resulting semantics is highly compositional, that is, denotations of compound phrases and sentences are created using function application, according to the syntactic structure of the query, and any phrase can be replaced by any other phrase of the same syntactic category.

```
produced tmph = [ subject | (subject, objects) <-
                                revers_image2 "producer",
                                tmph objects ]
```

Listing 15: Transitive verb: *produced*

Intransitive verbs

Intransitive verbs, unlike transitive verbs, do not have an object or phrase receiving the action. Furthermore, a intransitive verb denotes a function in which the set of entities that are subjects of the type associated with that verb. For example, the intransitive verb *produce* as in "Brad Pitt produce" will return all subjects that have an object URI that represents *producer* in DBpedia. In other words, the definition of *produce* is defined below.

```
--Intransitive verbs
--returns all subjects that are producers
produce = getts_1 ("?", type0, yago "AmericanFilmProducers")
--returns all subjects that are directors
direct = getts_1 ("?", type0, yago "AmericanFilmDirectors")
```

Listing 16: Intransitive verbs: *produce and direct*

The denotation of intransitive verbs help to answer queries in which phrases include verbs that do not require an object or a term phrase.

Example queries

In this section, sample queries are evaluated and the steps in which they are evaluated is explained. The answer to a query is mathematically composed from the meaning of its parts. The syntax of the query creates the order of evaluation.

The composition of the meaning of the query, and parts of it, is determined by a formal denotational semantics.

The query "Brad Pitt is a director" is considered and evaluated as follows.

```
--Brad Pitt directs
brad_pitt directs = checkMember (res "Brad_Pitt") directs
  = checkMember (dbpedia:Brad_Pitt) [ dbpedia:Chris_Sparling,
        dbpedia:Aram_Avakian, dbpedia:Barnet_Kellman,
        dbpedia:Bobby_Burns, dbpedia:Chris_Sanders_(director),
        dbpedia:Christopher_Smith_(performer),
        dbpedia:Dana_Adam_Shapiro, dbpedia:David_Duchovny,...]
  = False
```

This query evaluates the proper noun for "Brad Pitt" that take a set of entities *directs* as an argument and which return True if a particular entity is a member of that set. The query result is noted as *False* since the resource URI which represents "Brad Pitt" is not a member of the set of entities denoted by "directs".

Secondly, the query "Clint Eastwood is a producer and a director" is considered and evaluated. After parsing, the query is translated to as follows.

```
--Clint Eastwood produces and directs
clint_eastwood (nounand produces directs)
 = checkMember (dbpedia:Clint_Eastwood) (nounand [
        dbpedia:Ali_LeRoi, dbpedia:Alina_Panova,
        dbpedia:Alvin_H._Perlmutter, dbpedia:Anthony_Cardoza,
        dbpedia:B._P._Schulberg, dbpedia:Bitsie_Tulloch,
        dbpedia:Bobby_Miller_(filmmaker),...,
        dbpedia:Clint_Eastwood,...] [ dbpedia:Chris_Sparling,
        dbpedia:Aram_Avakian, dbpedia:Barnet_Kellman,
        dbpedia:Bobby_Burns, dbpedia:Chris_Sanders_(director),
        dbpedia:Christopher_Smith_(performer),
        dbpedia:Dana_Adam_Shapiro, dbpedia:David_Duchovny,...,
        dbpedia:Clint_Eastwood,...])
```

The "nounand" function intersects the two lists to create a single 'common' list, only entities that are common in both input lists will appear in the single common list.

```
= checkMember (dbpedia:Clint_Eastwood) [ dbpedia:Ali_LeRoi,
      dbpedia:Alina_Panova, dbpedia:Alvin_H._Perlmutter,
      dbpedia:Anthony_Cardoza, dbpedia:B._P._Schulberg,
      dbpedia:Bitsie_Tulloch, dbpedia:Bobby_Miller_(filmmaker),
      dbpedia:Chris_Sparling, dbpedia:Aram_Avakian,
      dbpedia:Barnet_Kellman, dbpedia:Bobby_Burns,
      dbpedia:Chris_Sanders_(director),
      dbpedia:Christopher_Smith_(performer),
      dbpedia:Dana_Adam_Shapiro, dbpedia:David_Duchovny,...,
      dbpedia:Clint_Eastwood,...])
= True
```

This query evaluates the proper noun for "Clint Eastwood" that take a set of entities, the single common result list from the "nounand" function, as an argument and which return True if a particular entity is a member of that set. The query result is noted as *True* since the resource URI which represents "Clint Eastwood" is a member of the set of entities denoted by "nounand produces directs".

The query "Clint Eastwood directed a film" is evaluated as follows:

```
--Clint Eastwood directed a film
clint_eastwood (directed (a film))
= checkMember dpedia:Clint_Eastwood [
      dbpedia:Aaron_Katz_(filmmaker), dbpedia:Abraham_Polonsky,
      dbpedia:Akiva_Schaffer, dbpedia:Al_Pacino,
      dbpedia:Alain_Zaloum, dbpedia:Alan_Crosland,
      dbpedia:Alan_Gibson_(director), dbpedia:Albert_Finney,
      dbpedia:Albert_Magnoli, ..., dpedia:Clint_Eastwood, ...]
= True
```

As explained in the Semantics, The "directed" function takes a proper noun

or a term phrase and creates a subject list which is properly associated with the predicate that is denoted by the actual name of the its argument.

The query result is noted as *True* since the resource URI which represents "Clint Eastwood" is a member of the set of entities denoted and evaluated by "*directed (a film)*".

The query "Clint Eastwood directed Million Dollar Baby" is evaluated as follows:

```
--Clint Eastwood directed Million Dollar Baby?
clint_eastwood (directed million_dollar_baby)
 = checkMember dbpedia:Clint_Eastwood [ dbpedia:Clint_Eastwood ]
 = True
```

The "directed" function takes a proper noun or a term phrase and creates a subject list which are properly connected by the predicate property *director* and the object or term phrase denoted by the actual name of the directed function's argument. In this case, the "directed" function will take the "million_dollar_baby" proper noun and create a list of subjects connected by predicate property *director* and returns the list of subjects that are associated with the *dbpedia* : *million_dollar_baby* proper noun.

The query result is noted as *True* since the resource URI which represents "Clint Eastwood" is a member of the set of entities denoted and evaluated by "*directed million_dollar_baby*". The function denoted by "clint_eastwood" returns True when applied to the list of entities who directed Million Dollar Baby.

3.2 Contribution to DEV-NLQ

The author of this thesis co-authored two papers [7] and [8], which were presented at EDBT/ICDT 2014 Joint Conference and ESWC 2014, respectively.

The author's major contribution to *An Event-Driven Approach for Querying Graph-Structured Data Using Natural Language* [7] was to design the event-based graph and the semantics in the Haskell programming language.

The author's major contribution to *A Demonstration of a Natural Language Query Interface to an Event-Based Semantic Web Triplestore* [8] was to design

the event-based graph and semantics in the Haskell programming language and to demonstrate that bracketed English queries can be used to query DBpedia triple-stores, and other triple-stores like DBpedia. Another contribution is investigating how to extract sets of event-based n-tuple data-stores such as YAGO2's data-store [11].

In the paper *An Event-Driven Approach for Querying Graph-Structured Data Using Natural Language* [7], the co-author's name is listed as the second of five authors. In the paper *A Demonstration of a Natural Language Query Interface to an Event-Based Semantic Web Triplestore* [8], the co-author's name is listed as the fourth of five authors.

Another research contribution is the creation of [7]'s denotational semantics and the definition of arbitrarily-nested quantifications for DBpedia triple-stores. The use of the HSPARQL [20] package that is used to access triples from DBpedia is also a contribution.

The major research contribution of this thesis work is the extension of the paper and research described in [8] to show that the idea developed by Frost et al. can be used to query DBpedia.

4. TIMING EVALUATION OF QUERIES

This chapter shows the time evaluation of some example queries. To evaluate, we will first look at the experiment/algorithm design and pick the appropriate queries made in that process. In this chapter we will also look at the procedure of the data collection and how it was designed.

4.1 Experiment design

When testing various queries, it was important to choose the right queries for the experiment. After testing multiple queries, it was discovered that the Virtuoso DBpedia interface has a 10000 max row limit on result sets [4]. This limit was set to prevent abuse of the DBpedia public server by crawlers. The limit was also imposed by the DBpedia SPARQL endpoint for performance reasons. A solution to this problem would have been to download the DBpedia data and run all queries locally. However, the virtual machines that run the speechweb2 server [9] have a finite amount of disk space and RAM memory allocated which are significantly less than the requirements for the DBpedia dump files and datasets.

The requirements for setting up the DBpedia data on a local machine include a powerful machine with at least 4 Cores and 32 GigaBytes of RAM for DBpedia only. The requirements also include at least 256 GigaBytes of free Hard Drive space for downloading and repacking the DBpedia datasets, as well as the growing database file when importing the triples in to a graph [10].

The goal of the experiment is to test if results are complete and returned in reasonable amount of time. Most importantly, this experiment also helps determine if the algorithms designed can be optimized.

The queries used in this experiment are listed in table 4.1 below.

Tab. 4.1: Queries used in the experiment

ID	**Haskell Query**	**English query**
1	film	list all films
2	actor	list all actors
3	directed (a film)	directed a film
4	clint_eastwood (directed (a film))	Clint Eastwood directed a film
5	clint_eastwood (directed million_dollar_baby)	Clint Eastwood directed Million Dollar Baby
6	brad_pitt (produced (a film))	Brad Pitt produced a film
7	produced eat_pray_love	who produced Eat Pray Love
8	brad_pitt (produced eat_pray_love)	Brad Pitt produced Eat Pray Love
9	brad_pitt directs	Brad Pitt directs
10	clint_eastwood directs	Clint Eastwood directs
11	clint_eastwood (nounand produces directs)	Clint Eastwood produces and directs
12	produced_by (a (directed (million_dollar_baby)))	list films produced by a director of Million Dollar Baby
13	a person (directed (a film))	A person directed a film

The Haskell queries were evaluated using 2 methods: The first method consisted of running all valid queries in the Haskell program interface using a command line entry. In order to determine the timing for a valid query, the built-in timer was enabled before executing the query. The simplest way to enable the timer was to run the *':set +s'* command in Haskell. This allowed users to see the execution time and memory usage of any valid query that is run. This is the Haskell Program part of the experiment.

The second evaluation method consisted of designing a web program that is able to interface and run the Haskell program. The web program address is located at `http://speechweb2.cs.uwindsor.ca/wale_demo`. The timer starts when the query starts and ends when the query results are displayed. This is the Web Program part of the experiment.

4.2 Experiment result

Each query is executed 10 times for the web program. The average times are record by noting all query times and dividing the total query time by 10 to get the averages.

The full results are listed in table 4.2.

Tab. 4.2: Experiment results

ID	**Haskell Query**	**Web Program Average (seconds)**	**Haskell Program Average (seconds)**
1	film	5	2.8
2	actor	5	1.1
3	directed (a film)	81	78.82
4	clint_eastwood (directed (a film))	44	39.83
5	clint_eastwood (directed million_dollar_baby)	7.2	4.25
6	brad_pitt (produced (a film))	18.3	10.95
7	produced eat_pray_love	7	4.67
8	brad_pitt (produced eat_pray_love)	7	4.3
9	brad_pitt directs	4	1.47
10	clint_eastwood directs	2	1
11	clint_eastwood (nounand produces directs)	5	2.2
12	produced_by (a (directed million_dollar_baby))	13.1	8.86
13	a person (directed (a film))	97.6	92.74

All 10 times for queries 3, 4, 5, 6, 12 and 13 are show in the table 4.3.

Tab. 4.3: Experiment results: timings

ID	Haskell Query	Web Program (seconds)	Haskell Program (seconds)
3	directed (a film)	83, 84, 81, 80, 85, 83, 77, 77, 79, 81	79.06, 78.72, 78.87, 79.29, 78.87, 78.54, 78.54, 78.93, 78.61, 78.74
4	clint_eastwood (directed (a film))	46, 46, 45, 44, 44, 45, 41, 45, 43, 41	40.05, 39.86, 39.69, 39.77, 39.86, 39.84, 39.81, 39.86, 39.83, 39.75
5	clint_eastwood (directed million_dollar_baby)	7, 7, 7, 7, 7, 8, 6, 8, 8, 7	4.38, 4.12, 4.39, 4.39, 4.13, 4.40, 3.95, 4.38, 4.42, 3.94
6	brad_pitt (produced (a film))	19, 18, 18, 18, 21, 17, 19, 17, 18, 18	11.00, 10.57, 11.31, 10.76, 10.55, 11.60, 10.99, 11.13, 10.99, 10.62
12	produced_by (a (directed million_dollar_baby))	13, 13, 13, 13, 13, 13, 13, 13, 14, 13	9.04, 9.02, 8.94, 9.14, 7.95, 8.56, 9.03, 9.00, 8.91, 9.03
13	a person (directed (a film))	99, 101, 101, 102, 97, 97, 95, 94, 97, 93	93.34, 92.87, 93.56, 92.87, 90.12, 93.54, 93.34, 92.12, 92.98, 92.66

It is noted that queries executed with the web program require a few more seconds to complete. This is because the web program opens a connection to the Haskell program, executes the query, and closes the connection.

It is also noted that queries involving transitive verbs can take some time to complete. This is because the functions for transitive verbs require a collection to map a subject to a list of objects. The function ***collection*** takes a list of pairs:

[(director1, film1), (director2, film2), (director3, film3), ...],

and converts them to a map such that each subject is paired with a list of objects:

[(director1, [film1, film2, film3, ...]), ...].

Consider the query "directed (a film)" below:

Fig. 4.1: Sample query: directed (a film)

```
directed (a film)
                = [ subject | (subject, objects) <-
                        reverse_image2 "director",
                        a film objects ]
```

This query first creates a list. Each item in the list contains a subject-objects pair ***[(director1, [film1, film2, film3, ...]), ...]***, and then evaluates "a film ***[film1, film2, film3, ...])***" for all objects list in the subject-objects pair. Note that the query "film" returns a set of films (see page 12) and "a film ***[film1, film2, film3, ...])***" returns True if the set film and the set ***[film1, film2, film3, ...])*** have at least a common element (see page 17).

This process requires a lot of processing time.

Consider the query "directed million_dollar_baby" below:

Fig. 4.2: Sample query: directed million_dollar_baby

```
directed million_dollar_baby = [ subj |
                    (subj, objs) <- reverse_image2 “director",
                        million_dollar_baby objs]
```

This query first creates a list. Each item in the list contains a subject-objects pair

[(director1, [film1, film2, film3, ...]), ...]

and then evaluates

million dollar baby ***[film1, film2, film3, ...]***

for all objects list in the subject-objects pair. Note that the query

million dollar baby ***[film1, film2, film3, ...]***

returns True if the URI entity representing million dollar baby is in the set ***[film1, film2, film3, ...]*** (see page 16).

This explains the timing results of the query "directed (a film)" and the query "directed million_dollar_baby". million_dollar_baby is single entity, while film is a set of at most 10000 entities (see page 41 for more on the limitation).

However, queries involving intransitive verbs only take a couple of seconds. This is because the transitive verbs definition does not need to make a collection, it only needs to return a list associated with that verb. Furthermore, the queries that involving proper nouns will only check for that resource URI (entity) in a list of entities. For example, the query

clint_eastwood directs

is a proper noun definition that will checks the resource URI for "Clint Eastwood" in the list of entities denoted by ***directs***.

The complexity analysis of the algorithms designed for the thesis such as the ***collection*** method for transitive verbs, ***checkMember*** method for proper nouns, and quantifiers are discussed in chapter 5.

5. COMPLEXITY ANALYSIS OF QUERIES

The complexity of the algorithms defined in this paper are discussed and analysed in this section. The worst/average case is considered.

Firstly, the function ***checkMember*** is considered. This function is used in the denotation of proper nouns.

```
--checks if the element 'x' is in the list 'ls'
--Complexity => O(n)
checkMember x ls = List.elem x ls
```

For the average case scenario, the algorithm is a linear search, then the number of times it takes will be $n-(n-i)$, where $i \in [1...n]$. Therefore, finding the correct element in the i^{th} position will take $\mathcal{O}(n-(n-i))$. Since $\mathcal{O}(n-(n-i)) \in \mathcal{O}(n)$, the average case complexity is $\mathcal{O}(n)$.

The function ***interset*** is considered. This function is used in the definition of quantifiers.

```
--Finds a common set/list given 2 sets/lists
--Complexity => O(n+m)
interset s t = List.intersect s t
```

For the best case scenario, the algorithm will stop and return empty set if either of list s or list t is empty, then best case complexity is $\mathcal{O}(1)$.

For the average and worst case scenarios, the implementation is similar to $Set.intersection$ [3] and uses an efficient hedge algorithm comparable with hedge-union, in which the first list is converted into a set A, this operation is $\mathcal{O}(n \log n)$ [3]. Then the algorithm looks for elements of the second list in the set A. Since lookup on a set is $\mathcal{O}(1)$, then for all elements in the second list, complexity is $\mathcal{O}(m)$. The elements of the result come from the first set A [3]. Therefore, the complexity of the ***interset*** function is $\mathcal{O}((n \log n) + m)$, where n

is the number of elements in the first set and m is the number of elements in the second set.

The function ***collect*** is considered. This function is used in the definition of transitive verbs.

```
--create a map of key-values [(a,[b, ...]), ...]
--from a list of pairs [(a,b), ...]
--COLLECTION (expensive function - O(n^2))
collect [] = []
collect ((x,y):t) = (x, y:[e2 |
                        (e1, e2) <- t,
                        e1 == x]) :
                        collect [(e1, e2) |
                                        (e1, e2) <- t,
                                        e1 /= x ]
```

Frost et al. [8] designed a collection algorithm to build a map from a list of key/value pairs. However, there was concern that the definitions of Frost et al.'s *collect* function, which is used in the denotations of transitive verbs, is very expensive and may not be used for very large database triple-stores like DBpedia [4] and MusicBrainz [16].

For the best case scenario, the function will stop and return empty if the input list is empty, then best case complexity is $\mathcal{O}(1)$.

For the average and worst case scenarios, the function takes each pair and checks for all other pairs in the list with the same keys and concatenates the values together. Consequentially, the checking of the all pairs in each iterations requires $\mathcal{O}(n)$ lookup. Since there are n elements and the total lookup time is $\mathcal{O}(n)$, then total complexity is $\mathcal{O}(n) * n \in \mathcal{O}(n^2)$.

In order for the ***collect*** function to work efficiently with a sample of larger triple-stores, then modifications would be made to the function. A modification of these semantic definitions is to use the Map library [2] functions such as *fromListWith* and *toList* in Haskell.

```
--COLLECTION (a lot less expensive function - O(nlogn))
convertKVsList ls = (Map.toList .
                        Map.fromListWith (++) .
                        map (\ (x,y) -> (x,[y]))) ls
collect = convertKVsList
```

In the newly modified version of the ***collect*** function from right to left, there are 3 operations that are applied to the list of pairs.

1. $map(\backslash(x, y) -> (x, [y]))$ applies the singly-list style to all values in the pairs such that each pair (x, y) becomes $(x, [y])$. Example:

   ```
   map (\ (x,y) -> (x,[y]))
           [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")]
    == [(5,["a"]), (5,["b"]), (3,["b"]), (3,["a"]), (5,["a"])]
   ```

   This operation takes $\mathcal{O}(1)$ time for a single pair, therefore, for a list of pairs, the complexity is $(\mathcal{O}(1) * n) \in \mathcal{O}(n)$, where n is the number pairs in the list.

2. Map.fromListWith $(++)$ builds a map from a list of key/value pairs with a combining function. However, the result is not a list, it is a *Map* element. According to [2], its complexity is $\mathcal{O}(n \log n)$. Example:

   ```
   fromListWith (++)
           [(5,["a"]), (5,["b"]), (3,["b"]), (3,["a"]), (5,["a"])]
     = fromList [(3, ["a","b"]), (5, ["a","b","a"])]
   ```

3. Map.toList converts a *fromList* Map element to a list of key/value pairs. According to [2], its complexity is $\mathcal{O}(n)$. Example:

   ```
   toList (fromList [(3, ["a","b"]), (5, ["a","b","a"])])
     = [(3, ["a","b"]), (5, ["a","b","a"])]
   ```

Combining all 3 functions, the combining complexity of ***collect*** function $= (\mathcal{O}(n) + \mathcal{O}(n \log n) + \mathcal{O}(n)) \in \mathcal{O}(n \log n)$. Therefore, the final complexity of the ***collect*** function is $\mathcal{O}(n \log n)$. The effect of improving complexity from $\mathcal{O}(n^2)$ to $\mathcal{O}(n \log n)$

is considerable. Consider 10,000 "x directed y triples" $(10,000^2)$ is 100,000,000 whereas 10,000 logbase2 10,000 is approximately 10,000 * 13, which is approx 780 times less than 100,000,000.

6. REDUCING COMPUTATIONAL COST

6.1 Redefine transitive verbs

A way to reduce the computational cost of the semantics described in this paper is rewrite the definitions for transitive verbs using general knowledge. The algorithm for transitive verbs may have to be redefined to remove redundancy. For example, the query "*directed (a film)*" and "*directed (two films)*" will check if the list of objects are films then return the subject who directed the film.

```
directed tmph = [ subject | (subject, objects) <-
                                reverse_image2 "director",
                                tmph objects]
```

The phrase "a film" is redundant in "directed a film" because the word "directed" is only ever used with films, therefore the transitive verbs could be redefined to remove the redundant parts. The "directed" could be defined with no term-phrase to return all entities which are subjects of triples with predicate "director".

```
--Reducing computational cost
--alternate definition for 'directed a film'
directed_a_film = [ subj | (subj, objs) <-
                                reverse_image2 "director"]
```

Listing 17: Transitive verb: *directed a film*

```
--Reducing computational cost
--alternate definition for 'directed 2 films'
directed_two_films = [ subj | (subj, objs) <-
                                reverse_image2 "director",
                                length objs == 2]
```

Listing 18: Transitive verb: *directed two films*

The use of this redefinition must be carried out by the parser, and would be would be difficult to implement.

6.2 New definition of COLLECT function

```
-- collection (a lot less expensive function - O(nlogn))
convertKVsList ls = (Map.toList .
                        Map.fromListWith (++) .
                        map (\ (x,y) -> (x,[y]))) ls
collect = convertKVsList
```

In the newly modified version of the ***collect*** function from right to left, there are 3 operations that are applied to the list of pairs.

1. $map(\backslash(x, y) -> (x, [y]))$ applies the singly-list style to all values in the pairs such that each pair (x, y) becomes $(x, [y])$. Example:

   ```
   map (\ (x,y) -> (x,[y]))
           [(5,"a"), (5,"b"), (3,"b"), (3,"a"), (5,"a")]
    == [(5,["a"]), (5,["b"]), (3,["b"]), (3,["a"]), (5,["a"])]
   ```

2. Map.fromListWith (++) builds a map from a list of key/value pairs with a combining function. However, the result is not a list, it is a *fromList* element. Example:

   ```
   fromListWith (++)
           [(5,["a"]), (5,["b"]), (3,["b"]), (3,["a"]), (5,["a"])]
     = fromList [(3, ["a","b"]), (5, ["a","b","a"])]
   ```

3. Map.toList converts a *fromList* Map element to a list of key/value pairs. Example:

   ```
   toList (fromList [(3, ["a","b"]), (5, ["a","b","a"])])
     = [(3, ["a","b"]), (5, ["a","b","a"])]
   ```

The most important function is "*Map.fromListWith*". This function creates a tree map.

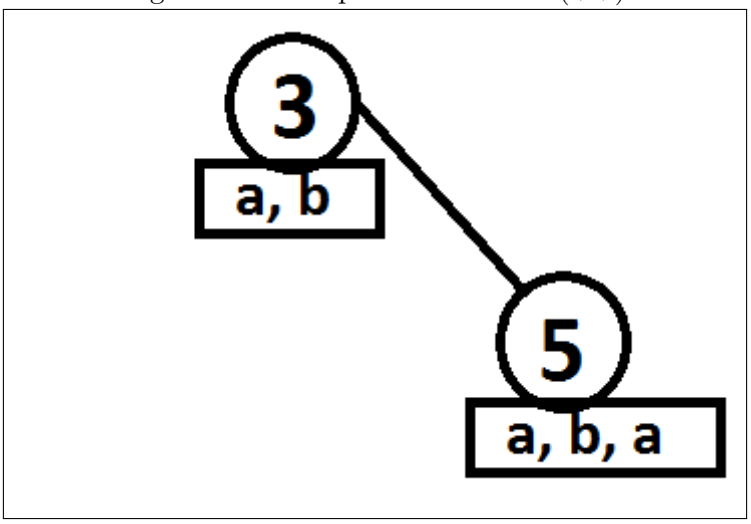
In a tree map, insertions and lookup require $\Theta(\log n)$ rotations if the tree map was maximally imbalanced before the element was inserted [2]. As a result, creating a tree-map from a list of pairs require $\mathcal{O}(n \log n)$ time, since $\mathcal{O}(\log n)$ time is required per each of the n pairs. If a duplicate key is found, the function that is passed, "++", is used to combine the values of those keys.

Consider the following list of pairs:

[(5,["a"]), (5,["b"]), (3,["b"]), (3,["a"]), (5,["a"])].

After converting the list into a Map using *fromListWith (++)*, we would get the map in Figure 6.1.

Fig. 6.1: TreeMap: fromListWith (++)



Inserting a pair into a balanced search tree is $\mathcal{O}(\log n)$; this is because inserting requires navigating through the tree node which leads to the location in the tree map in which the new value is. Since there are n elements to insert, there are exactly $\mathcal{O}(n \log n)$ bounds that the '*Map.fromListWith*' function has.

7. EXTENDING THIS APPROACH TO ACCOMMODATE PREPOSITIONAL PHRASES

According to Frost et al. [7] [8], complex prepositional phrases, such as "in 2004 or in 2004 in a city in California" have typically been somewhat difficult to integrate into a compositional NL query semantics which allows arbitrarily-nested quantification. A way to extend this approach to accommodate prepositional phrases is make use of YAGO2's [11] extended 5-tuples which allows the use a technique called reification to represent contextual data. Furthermore, extending this approach to accommodate prepositional phrases may require modifying the definition of transitive verbs. The definition of each transitive verb is redefined to make use of a list to filter entities which are in the image list of the transitive verb.

A recursive function called "filter" applies each prepositional phrase in turn as a filter to each entity.

```
--modified transitive verb "directed" denotation with
 ↪  prepositional phrase
directed1 tmph preps = [ subj| (subj, objs) <-
                         reverse_image2 "director",
                         tmph objs &&
                         length (filter1 objs preps) /= 0]
```

The "filter1" function will return the list of objects that satisfies all the prepositional conditions *preps*.

```
--filter for prepositional phrase
filter1 objs preps = [x | x <- objs,
                         containfilter x preps]
containfilter x [] = True
containfilter x (y:ys) = List.isInfixOf y x &&
                                  containfilter x ys

--a part of a prepositional phrase examples:
--"in 2008", "in 2010" and "in 2013"
in_2008 = ["2008"]
in_2010 = ["2010"]
in_2013 = ["2013"]
```

Consider the query "Did Ron Howard direct Rush in 2013"? This query converted to bracketed query becomes "ron_howard (directed1 rush in_2013)". This query, which includes prepositional phrase, can then be evaluated.

This concept is reliant on the fact that some URIs in DBpedia have representation of time. For example, the URI for the film 'Rush' is '`http://dbpedia.org/resource/Rush_(2013_film)`' or 'dbpedia:Rush_(2013_film)'.

Note that this idea is only a concept. Frost et al. [8] plan to improve and extend the semantics to accommodate prepositional phrases. A possible way forward is to use Frost et al.'s [8] event-based approach to develop a formal denotational semantics for the YAGO2 [11] data-store which may cover a wide range of NL constructs including complex chained prepositional phrases.

8. CONCLUSION

8.1 Proof of thesis

According to the experiment conducted in this thesis, it is proven that NLQIs designed where words and phrases are evaluated as expressions in lambda calculus can be used to query large datasets such as DBpedia.

It is possible to construct a usable, wide coverage natural language query interface to DBpedia triple-store by treating words and phrases in the query as expressions of the lambda calculus which are applied to each other according to the syntactic structure of the query. In this thesis, Frost et al.'s direct-evaluation approach has been adapted for use with DBpedia and have tested the revised approach with sample queries.

By *usable*, it is possible for answers to be returned in less than 100 seconds. The example queries presented in this thesis can all be evaluated in less than 100 seconds.

By *wide coverage*, the queries can include common nouns, proper nouns, quantifiers, intransitive and transitive verbs, and nested quantification. The queries presented in this thesis contain examples of all of these constructs.

By *extensible*, the approach can be extended for use with more powerful triple-stores, such as YAGO2 and event-based triple-stores. The approach described in this thesis regarding prepositional phrases can be used as a concept into further research.

8.2 Limitations

The proposed approach is able to answer many questions correctly. There is a limit in the result set returned by querying Dbpedia. Querying DBpedia will limit the result-set to 10000. For example, there are 60000+ films listed in DBpedia dataset, however, querying DBpedia for all films will only return the first 10000 results.

A solution is to use SPARQL sub-queries, such that a query uses a combination of multiple queries with limits and offsets. For example, querying DBPedia for all

films requires the following SPARQL query:

```
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
  } LIMIT 10000 OFFSET 0
}
UNION
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
  } LIMIT 10000 OFFSET 10000
}
UNION
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
  } LIMIT 10000 OFFSET 20000
}
UNION
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
  } LIMIT 10000 OFFSET 30000
}
UNION
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
```

```
  } LIMIT 10000 OFFSET 40000
}
UNION
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
  } LIMIT 10000 OFFSET 50000
}
UNION
{
 SELECT DISTINCT ?film_title
 WHERE {
    ?film_title rdf:type <http://dbpedia.org/ontology/Film> .
  } LIMIT 10000 OFFSET 60000
}
```

However, it would be difficult to translate the SPARQL query to Haskell using HSPARQL, and the total result-set count will have to be determined before all queries are evaluated.

8.3 Conclusion and future work

In this thesis, the event based approach that has been proposed by Frost et al. [7, 8] has been modified to query the DBpedia triple-store. We have also discussed how the approach could be extended to accommodate prepositional phrases if the triple-stores could represent contextual data.

Triple-stores are being developed to accommodate more informative contextual data such as YAGO2 [11] which uses a simple form of reification to represent temporal and spatial properties. Despite the fact that there are no available endpoints, that the author knows of, in which it is possible to query the YAGO2 data-store, Frost et al.'s research group are developing another denotational semantics so that NL queries can be evaluated directly with respect to YAGO2 data.

REFERENCES

[1] 2002, D. L. (2002a). Hackage: Data.List. `https://hackage.haskell.org/package/base-4.7.0.1/docs/Data-List.html`. [Online; accessed 19-July-2015].

[2] 2002, D. L. (2002b). Hackage: Data.Map. `https://hackage.haskell.org/package/containers-0.4.0.0/docs/Data-Map.html`. [Online; accessed 30-June-2015].

[3] 2002, D. L. (2002c). Hackage: Data.Set. `http://hackage.haskell.org/package/containers-0.5.6.3/docs/Data-Set.html`. [Online; accessed 30-June-2015].

[4] Auer, S., Bizer, C., Kobilarov, G., Lehmann, J., Cyganiak, R., and Ives, Z. (2007). *DBpedia: A Nucleus for a Web of Open Data*. Springer.

[5] Damljanovic, D., Agatonovic, M., and Cunningham, H. (2012). Freya: An interactive way of querying linked data using natural language. In *Proceedings of the 8th International Conference on The Semantic Web*, ESWC'11, pages 125–138, Berlin, Heidelberg. Springer-Verlag.

[6] Dowty, D. R., Wall, R., and Peters, S. (1981). *Introduction to Montague semantics*, volume 11. Springer.

[7] Frost, R. A., Agboola, W., Matthews, E., and Donais, J. (2014a). An event-driven approach for querying graph-structured data using natural language. In *Querying Graph Structured Data (GraphQ)), Organization=EDBT/ICDT 2014 Joint Conference, pages=192–199.*

[8] Frost, R. A., Donais, J., Matthews, E., Agboola, W., and Stewart, R. (2014b). A demonstration of a natural language query interface to an event-based semantic web triplestore. In *The Semantic Web: ESWC 2014 Satellite Events*, pages 343–348. Springer International Publishing.

[9] Hafiz, R. and Frost, R. A. (2010). Lazy combinators for executable specifications of general attribute grammars. In *Practical Aspects of Declarative Languages*, pages 167–182. Springer.

[10] Hees, J. (2014). Setting up a local DBpedia 2014 mirror with Virtuoso 7.1.0. `https://joernhees.de/blog/2014/11/10/setting-up-a-local-dbpedia-2014-mirror-with-virtuoso-7-1-0/`. [Online; accessed 27-June-2015].

[11] Hoffart, J., Suchanek, F. M., Berberich, K., and Weikuma, G. (2013). Yago2: A spatially and temporally enhanced knowledge base from wikipedia. *Artificial Intelligence*, 194:28–61.

[12] Höffner, K., Unger, C., Bühmann, L., Lehmann, J., Ngonga Ngomo, A.-C., Gerber, D., and Cimiano, P. (2013). User interface for a template based question answering system. In Klinov, P. and Mouromtsev, D., editors, *Knowledge Engineering and the Semantic Web*, volume 394 of *Communications in Computer and Information Science*, pages 258–264. Springer Berlin Heidelberg.

[13] Lehmann, J., Furche, T., Grasso, G., Ngomo, A.-C. N., Schallhart, C., Sellers, A., Unger, C., Bühmann, L., Gerber, D., Höffner, K., Liu, D., and Auer, S. (2012). Deqa: Deep web extraction for question answering. In *Proceedings of the 11th International Conference on The Semantic Web - Volume Part II*, ISWC'12, pages 131–147. Springer-Verlag, Berlin, Heidelberg.

[14] Lopez, V., Fernández, M., Motta, E., and Stieler, N. (2012). Poweraqua: Supporting users in querying and exploring the semantic web. *Semantic web*, 3(3):249–265.

[15] P(é)rez, J., Arenas, M., and Gutierrez, C. (2006). Semantics and complexity of sparql. In *The Semantic Web - ISWC 2006*, volume 4273, pages 30–43. Springer.

[16] Swartz, A. (2002). Musicbrainz: A semantic web service. *Intelligent Systems, IEEE*, 17(1):76–77.

[17] Tablan, V., Damljanovic, D., and Bontcheva, K. (2008). A natural language query interface to structured information. In *Proceedings of the 5th European*

Semantic Web Conference on The Semantic Web: Research and Applications, ESWC'08, pages 361–375. Springer-Verlag, Berlin, Heidelberg.

[18] Unger, C., Bühmann, L., Lehmann, J., Ngonga Ngomo, A.-C., Gerber, D., and Cimiano, P. (2012). Template-based question answering over rdf data. In *Proceedings of the 21st International Conference on World Wide Web*, WWW '12, pages 639–648, New York, NY, USA. ACM.

[19] Wan, Z., Zhai, Y., and Yu, H. (2013). A semantic web information integration system based on ontology-mapping and multilevel query interface. In Qi, G., Tang, J., Du, J., Pan, J., and Yu, Y., editors, *Linked Data and Knowledge Graph*, volume 406 of *Communications in Computer and Information Science*, pages 76–89. Springer Berlin Heidelberg.

[20] Wheeler, J. and Stewart, R. (2014). The hsparql package. http://hackage.haskell.org/package/hsparql-0.2.5 Author: Jeff Wheeler, Maintained by: Rob Stewart.

Appendices

PROGRAM SOURCE CODE

A.1 Filename: gangster_v7.hs

```
import qualified Data.List as List
import qualified Data.Tuple as Tuple
import qualified Data.Set as Set
import qualified Data.Map as Map
import qualified Data.HashSet as HashSet
import qualified Text.Regex.Posix as Posix

import Getts_v7

namespace_prop = "http://dbpedia.org/property/"
namespace_res = "http://dbpedia.org/resource/"
namespace_ont = "http://dbpedia.org/ontology/"
namespace_ctgry = "http://dbpedia.org/resource/Category:"
namespace_umbel = "http://umbel.org/umbel/rc/"
namespace_yago = "http://dbpedia.org/class/yago/"

type0 = "http://www.w3.org/1999/02/22-rdf-syntax-ns#type"
subject = "http://purl.org/dc/terms/subject"

prop fragment = namespace_prop ++ fragment
res fragment = namespace_res ++ fragment
ont fragment = namespace_ont ++ fragment
ctgry fragment = namespace_ctgry ++ fragment
umbel fragment = namespace_umbel ++ fragment
yago fragment = namespace_yago ++ fragment

count setofents = length(setofents)

person = getts_1("?", type0, ont "Person")
```

```
firsts =  map first
first (a,b) = a

seconds = map second
second (a,b) = b

thirdswithfirsts = map thirdwithfirst
thirdwithfirst (a,b,c) = (c,a)

thirds = map third
third (a,b,c) = c

invert = map Tuple.swap

-- |film
--film = getts_1 ("?", subject, ctgry "English-language_films")
--film = getts_1 ("?", type0, yago "English-languageFilms")
film = getts_1 ("?", type0, ont "Film")
--film = getts_1 ("?", subject, ctgry "English-language_films")
-- ++ getts_1 ("?", type0, yago "English-languageFilms")
-- ++ getts_1 ("?", type0, ont "Film")
films = film

-- |tv_show
--tv_show = getts_1 ("?", type0, ont "TelevisionShow")
tv_show = getts_1 ("?", subject, ctgry
 ↪  "English-language_television_programming")
tv_shows = tv_show

-- |actors
american_film_actor = getts_1 ("?", type0, yago
 ↪  "AmericanFilmActors")
```

```
english_film_actor = getts_1 ("?", type0, yago "EnglishFilmActors")
american_tv_actor = getts_1 ("?", type0, yago
 ↪  "AmericanTelevisionActors")
english_tv_actor = getts_1 ("?", type0, yago
 ↪  "EnglishTelevisionActors")
tv_actor = american_tv_actor ++ english_tv_actor
film_actor = american_film_actor ++ english_film_actor
--act = getts_1 ("?", type0, umbel "Actor") ++ film_actor ++
 ↪  tv_actor
act = american_film_actor
acts = act
actor = acts

-- /intransitive verbs
-- /producer
american_film_producer = getts_1 ("?", type0, yago
 ↪  "AmericanFilmProducers")
english_film_producer = getts_1 ("?", type0, yago
 ↪  "EnglishFilmProducers")
film_producer = american_film_producer ++ english_film_producer
american_tv_producer = getts_1 ("?", type0, yago
 ↪  "AmericanTelevisionProducers")
english_tv_producer = getts_1 ("?", type0, yago
 ↪  "EnglishTelevisionProducers")
tv_producer = american_tv_producer ++ english_tv_producer
produce = film_producer ++ tv_producer
produces = produce

-- /director
american_film_director = getts_1 ("?", type0, yago
 ↪  "AmericanFilmDirectors")
english_film_director = getts_1 ("?", type0, yago
 ↪  "EnglishFilmDirectors")
```

```
film_director = american_film_director ++ english_film_director
american_tv_director = getts_1 ("?", type0, yago
 ↪  "AmericanTelevisionDirectors")
english_tv_director = getts_1 ("?", type0, yago
 ↪  "EnglishTelevisionDirectors")
tv_director = american_tv_director ++ english_tv_director
direct = film_director ++ tv_director
directs = direct

-- /US_presidents
united_states_president = getts_1 ("?", subject, ctgry
 ↪  "Presidents_of_the_United_States")

-- /every noun is defined
-- /proper nouns
brad_pitt setofents = checkMember (res "Brad_Pitt") setofents
michelle_obama setofents = checkMember (res "Michelle_Obama")
 ↪  setofents
angelina_jolie setofents = checkMember (res "Angelina_Jolie")
 ↪  setofents
barack_obama setofents = checkMember (res "Barack_Obama") setofents
chuck_lorre setofents =  checkMember (res "Chuck_Lorre") setofents
quentin_tarantino setofents = checkMember (res "Quentin_Tarantino")
 ↪  setofents
django_unchained setofents = checkMember (res "Django_Unchained")
 ↪  setofents
eat_pray_love setofents = checkMember (res "Eat_Pray_Love")
 ↪  setofents
clint_eastwood setofents = checkMember (res "Clint_Eastwood")
 ↪  setofents
million_dollar_baby setofents =  checkMember (res
 ↪  "Million_Dollar_Baby") setofents
```

```
pulp_fiction setofents =  checkMember (res "Pulp_Fiction")
 ↪  setofents
guy_ritchie setofents = checkMember (res "Guy_Ritchie") setofents
julia_roberts setofents = checkMember (res "Julia_Roberts")
 ↪  setofents
mark_gatiss setofents = checkMember (res "Mark_Gatiss") setofents
steven_moffat setofents = checkMember (res "Steven_Moffat")
 ↪  setofents
benedict_cumberbatch setofents = checkMember (res
 ↪  "Benedict_Cumberbatch") setofents
kevin_feige setofents = checkMember (res "Kevin_Feige") setofents
the_godfather setofents = checkMember (res "The_Godfather")
 ↪  setofents
tyler_perry setofents = checkMember (res "Tyler_Perry") setofents
jodie_foster setofents = checkMember (res "Jodie_Foster") setofents
cowboys_and_aliens setofents = checkMember (res "Cowboys_&_Aliens")
 ↪  setofents
ron_howard setofents = checkMember (res "Ron_Howard") setofents
angels_and_demons setofents = checkMember (res "Angels_&_Demons")
 ↪  setofents
j_edgar setofents = checkMember (res "J._Edgar") setofents
rush setofents = checkMember (res "Rush_(2013_film)") setofents

-- |checks if the element 'x' is in the list 'ls'
-- |Complexity => O(n)
checkMember x ls = List.elem x ls

-- |Complexity => O(nlogn)
makeset x = Set.toList (Set.fromList x)
that s t = nounand s t
nounand s t = interset s t
nounor s t = makeset(s ++ t)
what x = x
```

```
did x = x

-- |Quantifiers
a nph vbph = length (interset nph vbph) /= 0
every nph vbph = subset nph vbph
one nph vbph = length(interset  nph vbph) == 1
two nph vbph = length(interset  nph vbph) == 2
which nph vbph = interset nph vbph
how_many nph vbph = length (interset nph vbph)
who vbph = which person vbph
was nph vbph = if nph vbph then "Yes" else "No"

-- |Finds a common set given 2 sets
-- |Complexity => O(n+m)
interset s t = List.intersect s t
--interset s t = (s List.\\ (s List.\\ t))

-- |Complexity = O(nlogn) + O(mlogm)
--interset s t = Set.toList (Set.intersection (Set.fromList s)
↪  (Set.fromList t))

-- |Complexity = O(nlogn) + O(mlogm)
subset s t = Set.fromList s `Set.isSubsetOf` Set.fromList t

-- |Image Relation
image2 rel = collect (getts_1_3 ("?", prop rel, "?"))

-- |Reverse Image Relation
reverse_image2 rel = collect (invert (getts_1_3 ("?", prop rel,
↪  "?")))

-- |collection (expensive function -> O(n^2))
--collect [] = []
```

```
--collect ((x,y):t) = (x, y:[e2 | (e1, e2) <- t, e1 == x]) :
↪  collect [(e1, e2) | (e1, e2) <- t, e1 /= x]

-- |collection (a lot less expensive function -> O(nlogn))
convertKVsList ls = (Map.toList . Map.fromListWith (++) . map (\
↪  (x,y) -> (x,[y]))) ls

collect = convertKVsList

{-
-- |Faster collect: runs in n lg n time
-- |Written by: Shane Peeler
collect = condense . sortFirst

-- |condense computes the image under a sorted relation
-- |condense runs in O(n) time and is lazy, also is lazy in the
↪  list computed in each tuple
condense :: (Eq a, Ord a) => [(a, a)] -> [(a, [a])]
condense [] = []
condense ((x,y):t) = (x, y:a):(condense r)
        where
        (a, r) = findall x t
        findall x [] = ([], [])
        findall x list@((t,y):ts) | x /= t = ([], list)
        findall x ((t,y):ts) | x == t = let (a2, t2) = (findall x
↪  ts) in (y:a2, t2)

-- |sort
sortFirst = List.sortBy (\x y -> compare (fst x) (fst y))
-}

-- |Transitive Verbs
-- |Produce
```

```
produced tmph = [ subj | (subj, objs) <- reverse_image2 "producer",
 ↪  tmph objs]
producer_of = produced
produced_by tmph = [ subj | (subj, objs) <- image2 "producer", tmph
 ↪  objs]

producer = produced (a film)

-- |Direct
directed tmph = [ subj | (subj, objs) <- reverse_image2 "director",
 ↪  tmph objs]
director_of = directed
directed_by tmph = [ subj | (subj, objs) <- image2 "director", tmph
 ↪  objs]

director = directed (a film)

-- |Reducing computational cost
-- |alternate definition for 'directed a film'
directed_a_film = [ subj | (subj, objs) <- reverse_image2
 ↪  "director"]
-- |alternate definition for 'directed 2 films'
directed_two_films = [ subj | (subj, objs) <- reverse_image2
 ↪  "director", length objs == 2]

-- |acted
acted_in tmph = [ subj | (subj, objs) <- reverse_image2 "starring",
 ↪  tmph objs]
starred_in = acted_in
movie_star = starred_in (a film)

-- |Transitional verb with prepositional phrase
directed1 tmph preps = [ subj| (subj, obj) <-
```

```
                        reverse_image2 "director",
                        tmph obj && length (filter1 obj preps) /=
                        ↪  0]

-- |filter for prepositional phrase
filter1 objs preps = [x | x <- objs, containfilter x preps]
containfilter x [] = True
containfilter x (y:ys) = List.isInfixOf y x && containfilter x ys

-- |a part of a prepositional phrase examples:
-- |"in 2008", "in 2010" and "in 2013"
in_2008 = ["2008"]
in_2010 = ["2010"]
in_2013 = ["2013"]

-- |marry/spouse
married tmph = [ subj | (subj, objs) <- image2 "spouse", tmph objs]
-- |born
born_in tmph = [ subj | (subj, objs) <- image2 "placeOfBirth", tmph
↪  objs]

-- |Sample Queries
-- |did Brad Pitt produce a film?
did_quentin_tarantino_direct_a_film = quentin_tarantino (directed
↪  (a film))

-- |did Clint Eastwood direct Million Dollar Baby
clint_eastwood_direct_million_dollar_baby = clint_eastwood
↪  (directed (million_dollar_baby))

-- |list films produced by the director of Million Dollar Baby
--produced_by (directed (million_dollar_baby))
```

```
-- |list films produced by Kevin Feige
--produced_by kevin_feige

-- |did Brad Pitt produce a film?
did_brad_pitt_produce_a_film = brad_pitt (produced (a film))
did_brad_pitt_produce_eat_pray_love = brad_pitt (produced
 ↪  (eat_pray_love))

-- |did Brad Pitt produce every film?
did_brad_pitt_produce_every_film = brad_pitt (produced (every
 ↪  film))

-- |Is Brad Pitt a Producer?
is_brad_pitt_a_producer = brad_pitt producer

-- |Is Brad Pitt an actor?
is_brad_pitt_an_actor = brad_pitt act

-- |Is Michelle Obama married to a US President?
did_michelle_obama_marry_a_US_president = michelle_obama (married
 ↪  (a united_states_president))

-- |Is Michelle Obama a US president?
is_michelle_obama_a_US_president = michelle_obama
 ↪  united_states_president

-- |Is Barack Obama a US president?
is_barack_obama_a_US_president = barack_obama
 ↪  united_states_president

-- |Is Chuck Lorre a producer?
is_chuck_lorre_a_producer = chuck_lorre producer
```

```
-- |OR / AND
termor tmph1 tmph2
  = f
    where f setofevs = (tmph1 setofevs) || (tmph2 setofevs)

termand tmph1 tmph2
  = f
    where f setofevs = (tmph1 setofevs) && (tmph2 setofevs)
```

A.2 Filename: getts_v7.hs

```
{-# LANGUAGE OverloadedStrings #-}

module Getts_v7
    (
    getts_1,
    getts_2,
    getts_3,
    getts_1_3,
    getts_1_2,
    getts_2_3
    )
where
import Control.Monad (forM_)
import Data.List (intersperse)
import Data.String.Unicode (unicodeRemoveNoneAscii)
import Data.String.Utils (split)
import Data.Set as Set

import Data.RDF hiding (triple)
import Database.HSparql.Connection
import Database.HSparql.QueryGenerator
--import Connection
```

```
--import QueryGenerator
import Data.Text hiding (head, concat, map)
import System.IO.Unsafe

--endpoint = "http://speechweb2.cs.uwindsor.ca/sparql"
endpoint = "http://dbpedia.org/sparql"
--endpoint = "http://live.dbpedia.org/sparql"

getts_1' :: (t, Text, Text) -> IO [[BindingValue]]
getts_1' (a, b, c) = do
     (Just s) <- selectQuery endpoint getts_1_query
     return s
     where
       getts_1_query = do
          x <- var
          triple x (iriRef b) (iriRef c)
          return SelectQuery { queryVars = [x] }


getts_2' :: (Text, Text, Text) -> IO [[BindingValue]]
getts_2' (a, b, c) = do
     (Just s) <- selectQuery endpoint getts_2_query
     return s
     where
       getts_2_query = do
          x <- var
          triple  (iriRef a) x (iriRef c)
          return SelectQuery { queryVars = [x] }

getts_3' :: (Text, Text, Text) -> IO [[BindingValue]]
getts_3' (a, b, c) = do
     (Just s) <- selectQuery endpoint getts_3_query
     return s
```

```
      where
        getts_3_query = do
           x <- var
           triple (iriRef a) (iriRef b) x
           return SelectQuery { queryVars = [x] }

getts_1_3' :: (Text, Text, Text) -> IO [[BindingValue]]
getts_1_3' (a, b, c) = do
      (Just s) <- selectQuery endpoint getts_1_3_query
      return s
      where
        getts_1_3_query = do
           x <- var
           z <- var
           triple x (iriRef b) z
           return SelectQuery { queryVars = [x, z] }

getts_2_3' :: (Text, Text, Text) -> IO [[BindingValue]]
getts_2_3' (a, b, c) = do
      (Just s) <- selectQuery endpoint getts_2_3_query
      return s
      where
        getts_2_3_query = do
           y <- var
           z <- var
           triple (iriRef a) y z
           return SelectQuery { queryVars = [y, z] }

getts_1_2' :: (Text, Text, Text) -> IO [[BindingValue]]
getts_1_2' (a, b, c) = do
      (Just s) <- selectQuery endpoint getts_1_2_query
      return s
      where
```

```
        getts_1_2_query = do
            x <- var
            y <- var
            triple x y (iriRef c)
            return SelectQuery { queryVars = [x, y] }

getts_1 :: (String, String, String) -> [String]
getts_1 (a, b, c) = preprocess (getts_1'(pack a, pack b, pack c))
getts_2 :: (String, String, String) -> [String]
getts_2 (a, b, c) = preprocess (getts_2'(pack a, pack b, pack c))
getts_3 :: (String, String, String) -> [String]
getts_3 (a, b, c) = preprocess (getts_3'(pack a, pack b, pack c))

getts_1_3 :: (String, String, String) -> [(String, String)]
getts_1_3 (a, b, c) = pair_up (preprocess (getts_1_3'(pack a, pack
 ↪  b, pack c)))
getts_2_3 :: (String, String, String) -> [(String, String)]
getts_2_3 (a, b, c) = pair_up (preprocess (getts_2_3'(pack a, pack
 ↪  b, pack c)))
getts_1_2 :: (String, String, String) -> [(String, String)]
getts_1_2 (a, b, c) = pair_up (preprocess (getts_1_2'(pack a, pack
 ↪  b, pack c)))

preprocess :: IO [[BindingValue]] -> [String]
preprocess = Prelude.map nodeToString . concat . dropDups .
 ↪  unsafeDupablePerformIO

-- Simple decode of RDF
nodeToString :: BindingValue -> String
nodeToString (Bound (UNode uriText)) = unpack (Prelude.last
 ↪  (splitOn "#" uriText))
nodeToString (Bound (LNode (TypedL text _))) = unpack text
nodeToString (Bound (LNode (PlainL text))) = unpack text
```

```
nodeToString (Bound (LNode (PlainLL text text2))) = unpack text
nodeToString (Bound other) = show other
nodeToString _ = "unknown"

--dropDups :: (Eq a) => [a] -> [a]
--dropDups = Set.toList . Set.fromList
--{-
dropDups :: (Eq a) => [a] -> [a]
dropDups [] = []
dropDups (x:xs) = if elem x xs
                  then dropDups xs
                  else x : dropDups xs
--}
pair_up :: [String] => [(String, String)]
pair_up [] = []
pair_up (x:y:ys) = (x, y) : pair_up ys
```

VITA AUCTORIS

Wale Agboola was born in 1989 in Ibadan, Nigeria. In 2007, he graduated from Kennedy Collegiate Institute in Windsor, Ontario. From there he went to the University of Windsor where he obtained a Bachelor of Science degree with Honours in Computer Science specializing in Software Engineering in 2011. In the Fall of 2015, he graduated with a Master of Science Degree in Computer Science from the University of Windsor in Ontario, Canada.