2016

Accommodating prepositional phrases in a highly modular natural language query interface to semantic web triplestores using a novel event-based denotational semantics for English and a set of functional parser combinators

Shane Peelar
University of Windsor

Accommodating prepositional phrases in a highly modular natural language query interface to semantic web triplestores using a novel event-based denotational semantics for English and a set of functional parser combinators

By:

Shane Peelar

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

Windsor, Ontario, Canada

2016

Accommodating prepositional phrases in a highly modular natural language query interface to semantic web triplestores using a novel event-based denotational semantics for English and a set of functional parser combinators

by

Shane Peelar

APPROVED BY:

Dr. Richard J. Caron
Department of Mathematics and Statistics

Dr. Luis G. Rueda
School of Computer Science

Dr. Robert D. Kent, Co-Supervisor
School of Computer Science

Dr. Richard A. Frost, Co-Supervisor
School of Computer Science

December 12 2016

Declaration of Co-Authorship / Previous Publication

I hereby declare that this thesis incorporates material that is result of joint research, as follows:

Some of the material in this thesis is derived from the following research papers:

[25] R. A. Frost, R. Hafiz, and P. Callaghan. "Parser combinators for ambiguous left-recursive grammars". In: *International Symposium on Practical Aspects of Declarative Languages*. Springer LNCS Volume 4902. 2008, pp. 167–181

[10] R. A. Frost, W. Agboola, E. Matthews, and J. A. Donais. "An Event-Driven Approach for Querying Graph-Structured Data Using Natural Language". In: *EDBT/ICDT Workshops*. Vol. 2014. 2014, pp. 192–199

[11] R. A. Frost, J. Donais, E. Mathews, W. Agboola, and R. Stewart. "A Demonstration of a Natural Language Query Interface to an Event-Based Semantic Web Triplestore". In: *ESWC (Satellite Events)*. Springer LNCS Volume 8798. 2014, pp. 343–348

Paper [25] (Frost, Hafiz, and Callaghan) describes a set of functional parser combinators developed by Frost and Hafiz as part of Hafiz's doctoral thesis work, which enables language processors to be built as executable specifications of fully-general attribute grammars, including ambiguous left-recursive grammars. The processors use a polynomial time complexity top-down parsing strategy which enables a natural specification of the grammars and the associated semantic rules. This was previously thought to be impossible, and was stated as such in many textbooks on parsing.

Paper [10] (Frost, Agboola, Matthews, and Donais) describes an event based semantics developed by Dr. Frost and his research team and includes extracts from a Haskell program which demonstrated the viability of the semantics with respect to an in-program database of triples coded as part of the program.

Paper [11] (Frost, Donais, Matthews, Agboola, and Stewart) describes the demonstration of the Haskell program which I wrote and which forms the basis of this thesis work.

The reason that I am not listed as an author is that the paper was submitted before I officially joined the research team. I developed the Haskell program after the paper was submitted. The online program was the one used by Dr. Frost in the demonstration he gave at the conference this paper was presented at.

My contributions to the research project include:

- Improving the efficiency of the programs which implement the event-based semantics
- Integrating the event-based semantics with the parser combinators to build the query processor
- Enhancing the existing module to access the external triplestore with efficient methods to do so, including a basic form of query fusion in the form of memoization
- Demonstrating a novel method of handling the word "by" in prepositional phrases, and extending prepositional phrases to span multiple property names
- Building a web interface to the query processor which includes both an English Natural Language Interface and also a safe Direct Query Interface for directly evaluating the combinators
- Converting the parser Hafiz wrote[25] to natively work with monads in Haskell, as well as the original semantics[11] to be monad based
- Maintaining the XSaiga package on *Hackage*[3], an online repository of Haskell libraries and programs, which contains the semantics, parser, and triplestore described in this Thesis

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.

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

The Semantic Web is an emerging component of the set of technologies that will be known as Web 3.0 in the future. With the large changes it brings to how information is stored and represented to users, there is a need to re-evaluate how this information can be queried. Specifically, there is a need for Natural Language Interfaces that allow users to easily query for information on the Semantic Web. While there has been previous work in this area, existing solutions suffer from the problem that they do not support prepositional phrases in queries (e.g, "in 1958" or "with a key"). To achieve this, we improve on an existing semantics for event-based triplestores that supports prepositional phrases and demonstrate a novel method of handling the word "by", treating it directly as a preposition in queries. We then show how this new semantics can be integrated with a parser constructed as an executable attribute grammar to create a highly modular and extensible Natural Language Interface to the Semantic Web that supports prepositional phrases in queries.

Dedication

This Thesis is dedicated to my meme and pepe, Theresa and Alfred Bombardier

Acknowledgements

I'd like to thank, in no particular order:

Dr. Richard A. Frost, whom I've been working with since my first semester here at the University of Windsor, for his guidance in my work and the many opportunities he has provided me over the years to participate in research. You were the first professor I had a chance to have a conversation with when I started my Undergraduate degree here in 2009, and ultimately it is our conversations during the labs of 60-100 that got me seriously interested in functional programming in the first place.

Dr. Robert D. Kent, for believing in me and providing me with many opportunities to contribute here at the University, and generally being a very positive role model in my life. It is in no small part our talks that have allowed me to refocus, get my priorities straight, and get my life back in order.

My External and Internal Readers, Dr. Richard J. Caron and Dr. Luis G. Rueda, for being available on such short notice to participate on my Thesis Committee.

David MacMillan, Bryan St. Amour and Paul Preney, for being great lab partners, being up for interesting conversations, and always being willing to lend an ear. Thank you also Bryan and Paul for letting me use your LaTeX styles and giving my first draft a read.

All of the secretaries in the School of Computer Science for their help during my degree. Whether it was getting me through scheduling nightmares or helping me get things done by the deadlines, your help has been incredibly invaluable to me and I sincerely thank you for your work.

My girlfriend, Taylor Tracey Kyryliuk, for being incredibly supportive of me both in life and in my work.

And my friend Kyle Iaquinta for taking the time to proofread my Thesis and providing

me with positive encouragement throughout my degree.

Contents

List of Figures

List of Appendices

Nomenclature

Chapter 1

Chapter 3

Chapter 5

Chapter 9

Chapter 1

Introduction

1.1 Motivation

Gone are the days where we could assume that a toaster was just a toaster, a refrigerator was just a refrigerator, or a kettle was just a kettle. Increasingly, Internet-aware "smart" devices are becoming the norm, silently replacing their ordinary counterparts with versions that contain microprocessors, sensors, network adapters, and other electronic hardware. These devices are what comprise the *Internet of Things*, a revolutionary change in how we interact with devices and appliances around us. One benefit of this change is that these devices are able to be operated remotely and through alternative interfaces, drastically improving accessibility for those who have disabilities. Natural Language Interfaces, in particular, could carry tremendous value to these users.

The Internet of Things, or *IoT*, is built on the *Semantic Web*, a body of open standards that allows these devices to "speak the same language" to one another, much like how the *World Wide Web* operates today[34]. If it were possible to design a framework for building Natural Language Interfaces to the Semantic Web, Natural Language Interfaces could in turn be designed for these IoT-enabled devices.

1.2 The Semantic Web

To begin understanding the Semantic Web, first some terminology must be introduced.

Definition 1 (Uniform Resource Identifier)**.** *"A compact sequence of characters that identifies an abstract or physical resource"[29]*

A Uniform Resource Identifier may also be referred to as a *URI*. A URI may take the form of a *name*, a *location*, or possibly both at once. A common example of a URI is an HTTP URL for a website. Another example would be the ISBN of a book, or a telephone number. URIs are a fundamental component of the Semantic Web, as it standardizes the notion of identification for the devices, or more generally speaking, *agents* that comprise it. Having a canonical name for a given object helps agents to refer to it, and in turn understand what one another are referring to when communicating.

Definition 2 (Triple)**.** *A 3-tuple that has the form* (subject, predicate, object)*, where* subject, predicate*, and* object *are Uniform Resource Identifiers[15].*

Fundamentally, the Semantic Web is a network of online databases that store facts in the form of *triples*. These *triples* compose the basic elements of the Resource Description Framework data model that underlies the Semantic Web, and databases that contain them are commonly referred to as *triplestores* or *RDF triplestores*. When we refer to the `subject`, `predicate`, or `object` of a triple, we are referring to the first, second, or third component of that triple, respectively.

Traditionally, triplestores describe facts in terms of *entities*.

Definition 3 (Entity)**.** *"A thing capable of an independent existence that can be uniquely identified"[8]*

We call triplestores whose information is organized around entities *entity-based triplestores*.

Definition 4 (Entity-based Triplestore)**.** *A triplestore where the* subject *and* object *of the triples contained within it refer only to* entities

In this Thesis Report, we use the syntax `(s, v, o)` to describe a triple. However, in practice, different encodings are used to represent triples. For instance, in *N-Triples* syntax[14], the triple `(s, v, o)` would be represented using the string obtained by substituting the URIs that `s`, `v`, and `o` represent directly into their corresponding placeholders "`$s`", "`$v`", and "`$o`" in the following string: "`<$s> <$v> <$o>.`". In our Haskell code,

we would represent this triple using the tuple `(ss, vs, os)`, where `ss`, `vs`, and `os` are, respectively, the URIs that `s`, `v`, and `o` represent mapped to the `String` type in the language.

Therefore, in this Thesis Report the triples we refer to are "abstract" triples rather than the actual concrete representations of those triples in practice.

1.3 The Problem

The SPARQL Protocol and RDF Query Language is an attempt to provide an SQL-like interface to the Semantic Web. Currently, it is the de-facto method of querying RDF triplestores. SPARQL queries form patterns that define restrictions on the elements of triples. Only triples matching the query pattern are returned in the result. Users submit queries to a *SPARQL endpoint* which in turn executes the query against a triplestore and returns the results.

SPARQL is not intended to be used directly by humans, however. User-friendly Semantic Web query interfaces provide higher level metaphors for interacting with RDF triplestores, improving accessibility. These query interfaces then transform these metaphors into corresponding SPARQL queries. One type of user-friendly interface that has seen use in the Semantic Web is the Natural Language Interface. These interfaces allow users to query Semantic Web triplestores using spoken or written Natural Language queries.

Querying the Semantic Web using Natural Language is an active area of research interest, as it allows users with little to no technical background to construct queries for RDF triplestores. This allows, for instance, health or police databases to be queried by professionals with minimal effort by the user. It also enhances accessibility of the Semantic Web for users who have disabilities.

One problem in developing Natural Language Interfaces, however, is that they must be expressive enough for users to comfortably use. Ideally, a wide coverage of Natural Language constructs should be supported so that users can directly express their intent without having to modify their queries to work around restrictions. As there have been several attempts to construct a Natural Language Interface for the Semantic Web, and research is ongoing, the problem of developing such a system is non-trivial. A summary of existing

approaches for Natural Language Interfaces to query the Semantic Web is given below:

1.4 Existing approaches

ORAKEL

An ontology-aware English interface to the Semantic Web based on Montague semantics[27]. ORAKEL parses sentences according to a provided grammar, and evaluates queries based on a compositional semantics. It supports quantification, negation, and conjunction in queries. ORAKEL directly attempts to convert the input query to a SPARQL query.

QuestIO

An ontology-aware English interface to the Semantic Web that is keyword oriented, attempting to match words against concepts to hone down queries[26] . Uses SeRQL[33], a query language similar to SPARQL, to form queries. Sentences are transformed into SeRQL queries by the use of a formal semantics.

AutoSPARQL

A supervised machine learning approach using English to query the Semantic Web[21]. Queries are provided in the form of keywords, which are used to construct query trees. These are then converted to SPARQL queries. It is a feedback oriented system in that the user is expected to be actively involved in refining subsequent results by selecting candidates from the returned set that best match what the user is looking for.

1.5 Shortcomings of previous approaches

While these approaches have seen success, they all share a shortcoming in that they do not allow for prepositional phrases in queries, and therefore have limited coverage of the English language.

1.6 New approach

The work presented in this thesis draws on two main concepts: executable attribute grammars[25] and event-based denotational semantics[10].

Executable attribute grammars are a natural way to implement Natural Language processors[38], and since they allow top-down rather than bottom-up parsing, they are highly modular[25]. This makes them well suited to the natural specification of semantic rules, since the meanings of terminals and rules in the grammar are able to be defined alongside those rules and terminals in the grammar itself. Additionally, it is possible to handle left-recursion and ambiguity directly in executable attribute grammars efficiently, allowing these grammars to witness sentences in natural language that are inherently ambiguous.

English denotational semantics were first described by Dr. Richard Montague in 1970[19]. Montague proposed denotations for English words using characteristic functions described in higher order logic. The presence of universal quantification in these characteristic functions made Montague's semantics difficult to implement in a computationally tractable way, however. Frost et al. in 1989 presented an improved version of Montague semantics called *FLMS*[39] that addressed this need. Frost's approach was to use sets instead of characteristic functions, making the semantics computationally tractable. In addition, Frost proposed a denotation for transitive verbs which was missing in Montague's semantics[16].

Frost et al. modified FLMS in 2013 to produce a new semantics called *EV-FLMS*[16], intended for use with event-based triplestores. This event-based denotational semantics operates on event-based triplestores[11] rather than entity-based triplestores. Entity-based triplestores describe entities and their relations to other entities, but a problem exists in how to add contextual information to a particular triple. In an event-based triplestore, triples describe events rather than entities, and information about entities and their relationships to one another may be gleaned from the events in which they occur. Additional information about an event may be added by simply adding a new triple to the triplestore.

As an example, the sentence "Jane bought a pencil" could be represented in an entity-based triplestore with the triple

```
(Jane, purchased, pencil_1)
```

In an event-based triplestore, the same sentence could be represented with three triples:

```
(event1, subject, Jane)
(event1, type, purchase)
(event1, object, pencil_1)
```

Additionally, other information about the event may be added as well, for example including the purchase price with the triple: `(event1, cost, $1)`, or perhaps the time *t* the transaction occurred with `(event1, time, t)`.

In this thesis, we present a new event-based denotational semantics called *Unified EV-FLMS* or *UEV-FLMS* that improves on EV-FLMS by unifying several semantic concepts, supporting prepositions that query multiple properties, and solving two problems resulting from how prepositional phrases were handled in the original semantics. In doing this, we introduce a novel method of handling the word "by" as used in the passive form of a verb by treating it directly as a preposition within our grammar, unifying our treatment of active and passive verbs. With this approach, we are able to accommodate queries such as "which moon was discovered in 1877 by hall" without any added complexity to the semantics.

1.7 Thesis Statement

By integrating a novel event-based denotational semantics with a parser constructed as an executable attribute grammar, it is possible to create a highly modular and extensible Natural Language Interface to the Semantic Web that supports the use of prepositional phrases in queries.

1.8 Proof of Concept

We prove the Thesis by creating an online English query interface to a triplestore containing thousands of facts about the solar system[2].

Some example queries that can be handled by this system include:

- "when was something discovered at mt_wilson"

- “how was the thing that was discovered at flagstaff discovered”
- “what was discovered in 1877 in us_naval_observatory”
- “what planet is orbited by a moon that was discovered in 1684”
- “which vacuumous moon that orbits jupiter was discovered by nicholson or hall with a telescope in 1938 in mt_wilson or mt_hopkins”

1.9 Structure of Thesis Report

The remainder of this Thesis Report is structured as follows:

1. Demonstration
2. The event-based semantics
3. The parser combinator
4. The query program
5. Timing
6. Thoughts on scaling up to handle massive triplestores
7. Proof of the Thesis
8. Conclusions

Chapter 2

Demonstration of the query interface that has been built

The query program is called "Solarman", and there exists two web based interfaces that can be used to interact with it.

Solarman was a program originally built in Miranda to demonstrate Frost's FLMS semantics in 1989[39], enabling the user to perform queries about objects in the Solar system. It was later ported to Haskell and integrated with Hafiz's parser in 2008[25] to form a Natural Language Interface using FLMS semantics to perform queries. Later still, it was ported to EV-FLMS semantics, operating on an in-program triplestore and, optionally, a SPARQL endpoint using `unsafeDupablePerformIO`. We ported Solarman to use UEV-FLMS semantics as described in this Thesis to perform queries on the Semantic Web through SPARQL endpoints safely, removing the need for `unsafeDupablePerformIO` in our code. We provide two interfaces: a Natural Language Interface enabling the user to enter queries in English, and a Direct Query Interface enabling users to enter queries using expressions in the Haskell language. Our SPARQL endpoint contains 4,129 triples in total representing information about the Solar system.

2.1 Natural Language Interface

The English Natural Language query interface can be accessed via this URL:

`http://speechweb2.cs.uwindsor.ca/solarman2/demo_sparql.html`

In the text box labeled "Enter query here", English queries about the Solar system can be entered to be evaluated. This is accomplished using the *Common Gateway Interface*, or *CGI*, to directly execute the "Solarman" program on the server with the given query as an argument. Internally, Solarman is configured to use a Virtuoso[23] RDF database as its SPARQL endpoint. The following SPARQL query is an example query that Solarman could send to a SPARQL endpoint:

```
PREFIX sol: <http://solarman.richard.myweb.cs.uwindsor.ca#>

SELECT DISTINCT ?x1 ?x0 WHERE {
?x0 sol:type sol:discover_ev .
?x0 sol:object ?x1 .
} ORDER BY ASC(?x1)
```

In the above example, the triplestore is being queried for all events of the "discover" type and the entities that were discovered in each of those events. Additionally, the query stipulates that the results should be lexicographically sorted in ascending order by the names of the entities being discovered in each event. This ordering constraint is useful as it enables the results to be processed into an FDBR in O(n) time, as detailed in chapter 5. Each pattern in a SPARQL query that defines restrictions on matching triples is described as a statement ending with a ".". As shown in the example, multiple pattern matching statements are permitted in a single SPARQL query. A "`PREFIX`" statement may be provided at the start of a SPARQL query to define a shorthand for a *namespace* that can be used within patterns. In our triplestore, all facts about the Solar system are in the "`http://solarman.richard.myweb.cs.uwindsor.ca#`" namespace.

Traversing the "More Examples" link will bring up a page that contains a full list of the words that can be used in queries, along with a list of example queries that can be used.

Some example queries that use prepositional phrases include:

- `what moon was discovered by one person in 1877`
- `what planet is orbited by a moon that was discovered in 1684`

- `what was discovered in 1877 in us_naval_observatory with a telescope`
- `how many moons were discovered in 1938`
- `where were the moons that were discovered by hall or kuiper in 1877 discovered`
- `which moons that orbit a planet that orbits a sun were discovered by one person at a place with a telescope`
- `how were the moons that were discovered with two telescopes discovered`
- `who discovered something with two telescopes`

The last query, "`who discovered something with two telescopes`", is a valid English sentence that is semantically ambiguous. The question could be asking whether someone used two telescopes to discover one particular thing, or whether someone discovered potentially multiple objects using two telescopes, using one telescope in each discovery. Our semantics treats both interpretations as valid and uses both in forming query results.

2.2 Direct Query Interface

In addition, a "Direct Query Interface" is provided that allows users to directly interact with the parser combinators and semantic functions by chaining them together to form queries. This is useful tool to explore and understand the semantics. It can be accessed via this URL:

`http://speechweb2.cs.uwindsor.ca/solarman2/demo_sparql_direct.html`

The Direct Query Interface is implemented using Safe Haskell[20], an extension of the Haskell language that restricts the functions that can be evaluated to a safe subset that is suitable for executing untrusted code. This makes the Direct Query Interface suitable for

use on public-facing websites, preventing external users from executing malicious code using the interface. For example, the expression `System.IO.readFile "/etc/passwd"` is disallowed under this scheme as both the Haskell `Prelude` and `System.IO` modules are by default not trusted.

Examples of Haskell expressions that can be executed with the Direct Query Interface:

- `when' $ something $ discovered_ [at mt_wilson]`
  (English: "when was something discovered at mt_wilson")
- `how' $ discovered $`
  `the (thing ‘that‘ discovered_ [at flagstaff])`
  (English: "how was the thing that was discovered at flagstaff discovered")
- `what $ discovered_ [in' 1877, at us_naval_observatory]`
  (English: "what was discovered in 1877 at us_naval_observatory")
- `which planet $ orbited_`
  `[by $ a (moon ‘that‘ discovered_ [in' 1684])]`
  (English: "what planet is orbited by a moon that was discovered in 1684")
- `which`
  `(liftM2 intersect_fdbr vacuumous (moon ‘that‘ orbits jupiter))`
  `$ discovered_ [by $ nicholson ‘termor‘ hall,`
  `with $ a telescope, in' 1938,`
  `at $ mt_wilson ‘termor‘ mt_hopkins]`
  (English: "which vacuumous moon that orbits jupiter was discovered by nicholson or hall with a telescope in 1938 in mt_wilson or mt_hopkins")

2.2.1 Verb voices

When using transitive verbs in the Direct Query Interface, the appropriate voice of the verb must be selected. Both the "discover" and "orbit" transitive verbs are supported. The available voices are summarized as follows:

- discover is the active voice (e.g. ”hall discovered a moon”)
- discover’ is the active voice with support for prepositional phrases, and
- discover_ is the passive voice with prepositional phrases

The above voices apply to the “orbit” verb as well.

2.2.2 Evaluating types

In addition to evaluating queries, the types of the combinators themselves and their results may also be evaluated by prepending the query with “`:t`”. For example:

- “`:t moon`” returns “`moon :: IO FDBR`”
- “`:t a moon`”

  returns “`a moon :: IO [(String, t2)] → IO [(String, t2)]`”
- “`:t by`” returns “`by :: t → ([[Char]], t)`”
- “`:t discovered’`”

  returns “`discovered’ :: (IO FDBR → IO FDBR) →`

  `[([[Char]], IO FDBR → IO FDBR)] → IO FDBR`”
- “`:t vacuumous`” returns “`vacuumous :: IO FDBR`”

In the above query results, `FDBR` refers to the “Function Defined By Relation” construct which is explained in Section 3.3.2. A value of type `IO FDBR` represents an “IO Action”, when executed, yields a value of type `FDBR`. In our semantics, the `FDBR` type is defined as `[(String, [String])]`. The type `a` → `b` is used to denote a function from type `a` to type `b`.

The Haskell language deduces the most general type declaration possible for a given function definition. If certain components of an input value aren’t used, those components will be replaced with a *type variable* when possible. This can be seen above in the queries for `:t a moon` and `:t by`. Neither of these expressions use the list of events associated

with the input FDBR, and therefore the type declaration was generalized, substituting a type variable for the list of events. The type variable is allowed to be bound to any type, subject to type constraints. For example, in "`:t a moon`", if we let `t2 = [String]`, then `a moon :: IO [(String, [String])]` and hence `a moon :: IO FDBR`.

2.2.3 Result formatting

The results of the Direct Query Interface are formatted for easier viewing. In particular, in each "FDBR", each result pair is on its own line, making it clear which entities are connected with which events.

2.3 XSaiga Package

"Solarman" is also included as part of the XSaiga package that we have uploaded to *Hackage*, an online repository of Haskell libraries and software[3]. It is available at this URL:

`https://hackage.haskell.org/package/XSaiga`

To install the XSaiga package, the GHC compiler version 8.0.1 or higher is required. With the *cabal* command, execute the following at a terminal:

```
> cabal update
> cabal install XSaiga
```

The XSaiga code resides inside the XSaiga namespace, and includes the parser, Solarman and its semantics, and a local triplestore in the module "LocalData" containing all of the RDF triples in a list.

2.4 Accessibility

Both interfaces are also designed to be accessible, supporting programs such as screen readers in order to accommodate those with disabilities. This was accomplished using the *WAVE Web Accessibility Evaluation Tool*[6] and *AChecker IDI Web Accessibility Checker*[1] to validate the interfaces for accessibility.

2.5 SPARQL Endpoint

The SPARQL endpoint that Solarman uses can be accessed via this URL:

`http://speechweb2.cs.uwindsor.ca/sparql`

We use the Virtuoso RDF triplestore software and its SPARQL interface[23] in our demonstration. To replicate our setup, first install the Virtuoso software for your operating system. Next, obtain a database dump of our triplestore in "n-Triples" format[14] using this URL:

`http://speechweb2.cs.uwindsor.ca/solarman2/triples.nt`

Finally, use the "bulk import" mechanism in Virtuoso Conductor to import our triples into your triplestore, selecting the `triples.nt` file. Our semantics can use any SPARQL interface, so the Virtuoso triplestore may be substituted with any number of alternatives, including Apache Jena[18].

2.6 Summary

In this chapter we summarized previous implementations of Solarman and described our online Natural Language Interface using our Semantics. In Chapter 3, we present an overview of event-based denotational semantics, summarizing EV-FLMS and introducing UEV-FLMS.

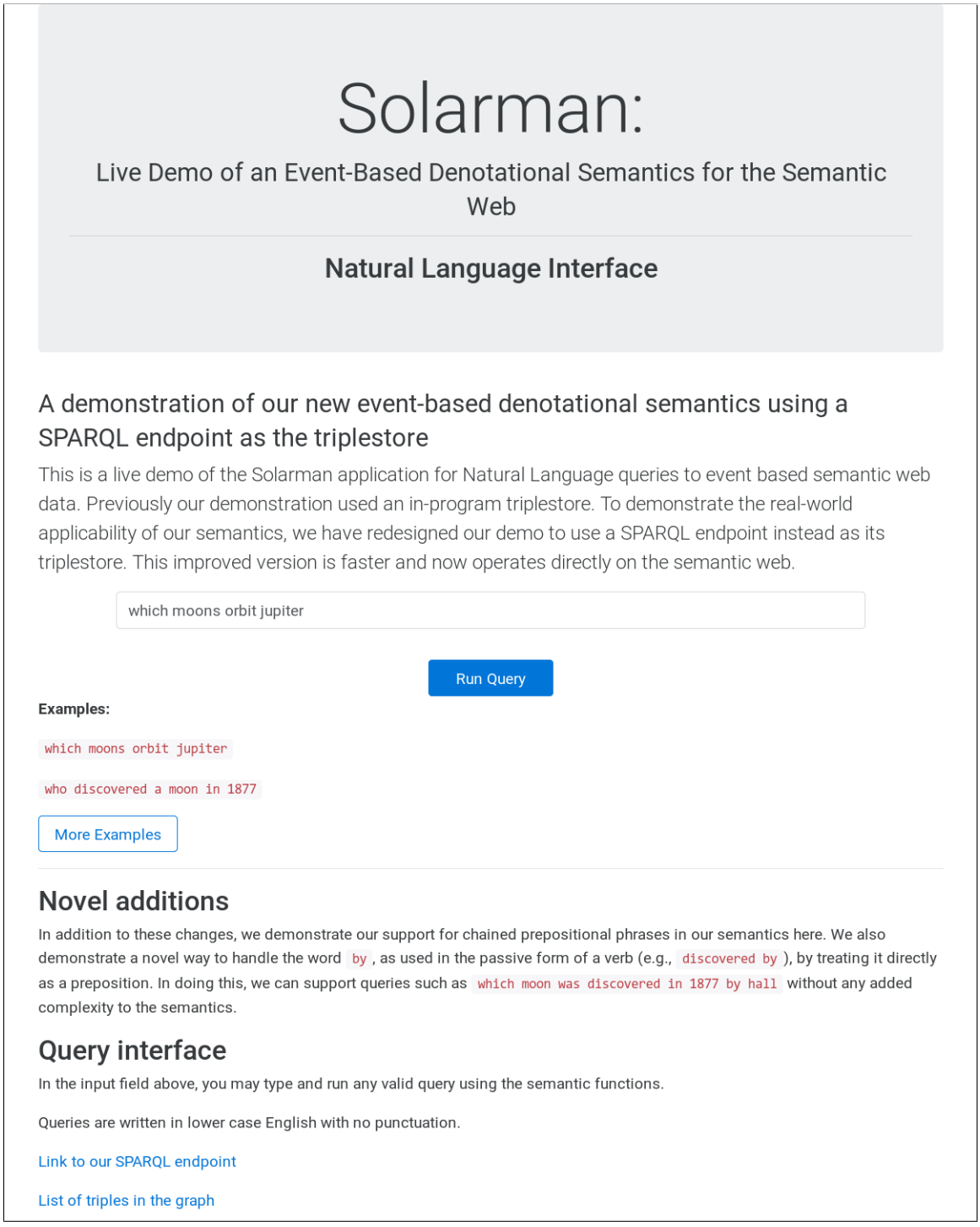


Figure 2.1: Screenshot of English Natural Language Interface

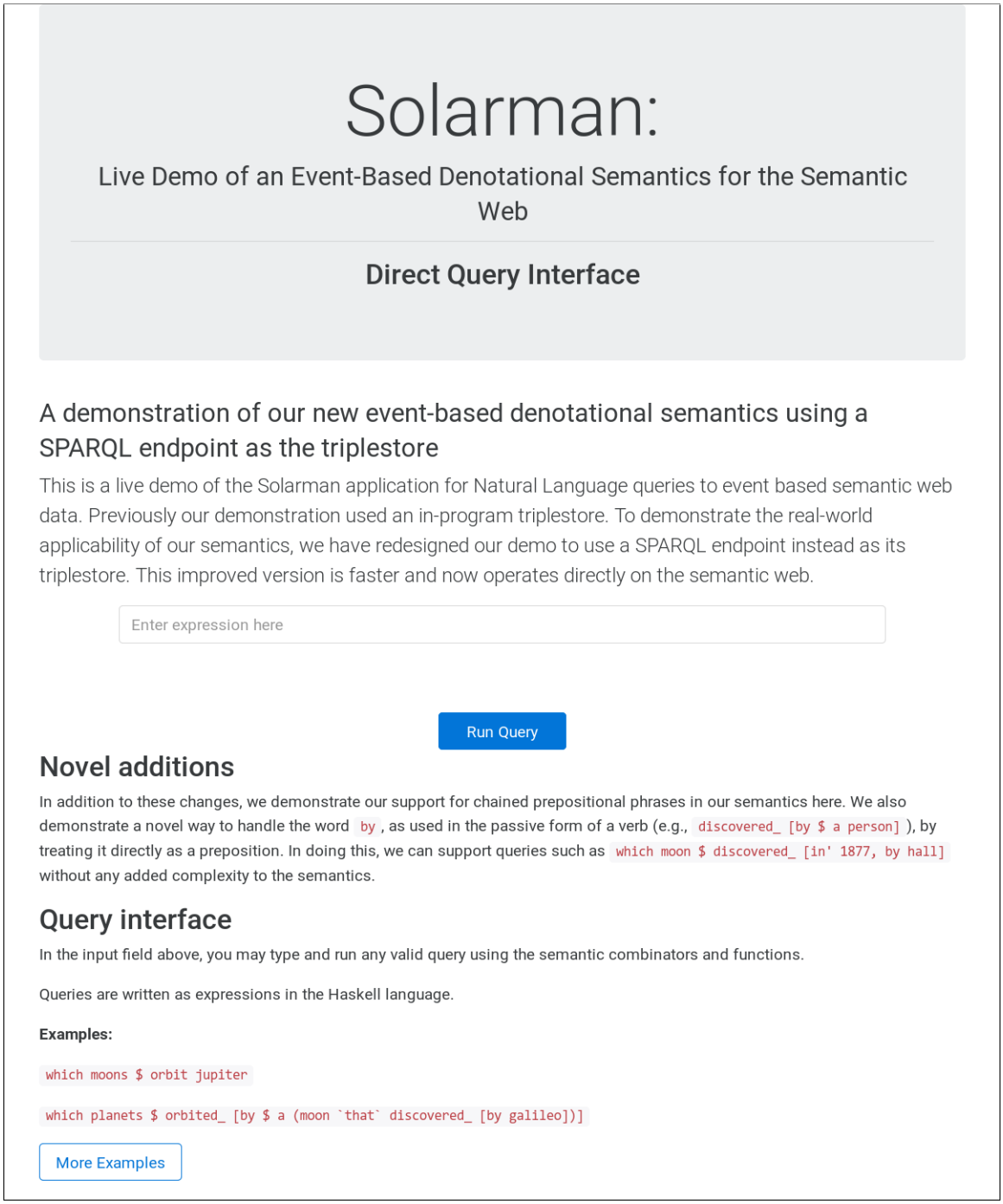


Figure 2.2: Screenshot of Direct Query Interface

Chapter 3

Event-Based Denotational Semantics

Our demonstration query program, Solarman, uses an event-based denotational semantics in order to perform queries on the Semantic Web. In this chapter we present an overview of event-based triplestores, a summary of the original event-based semantics that our semantics is based on, and our improved semantics. We also present a novel method of handling the word "by", as in the phrase "discovered by", treating it directly as a prepositional phrase in queries.

3.1 Event-Based Triplestores

One problem with entity-based triplestores is that it is difficult to add contextual information to a triple. Two common examples of contextual information are time and location. Many approaches that allow this use a method called *reification*[30].

One form of reification is to organize information into *events*.

Definition 5 (Event)**.** *A set of meaningfully connected physical or abstract phenomena*

For example, the triple `(sally, met, susan)` in an entity-based triplestore could be represented by three triples:

```
(event1, type, meet)
(event1, subject, sally)
(event1, object, susan)
```

These triples describe the event in which "Sally met Susan" rather than directly describing the meeting itself. The advantage of this approach is that it is possible to add additional information about the meeting by simply adding more triples with `event1` as the *subject*:

```
(event1, year, 1955)
(event1, location, windsor)
```

Triplestores that organize their information in this fashion are called *Event-based Triplestores*.

Definition 6 (Event-Based Triplestore)**.** *A triplestore where the* `subject` *of the triples contained within it refer to events[10]*

We say that a triple *belongs* to an event if the `subject` of the triple is the name of the event. For example, the triples above belong to `event1`. The entities that belong to an event `e` are the `objects` of the triples belonging to that event. For example, in the triples above, the entities `sally`, `susan`, `1955`, and `windsor` belong to `event1`.

The key motivation behind using Event-based triplestores in this Thesis is that they directly support reification on triples[10].

3.2 Original Event-Based Denotational Semantics

The semantics in this Thesis is based on work that was originally described by Frost et al. in 2013[16], called EV-FLMS. That work was later improved upon by Frost and Agboola in 2014[10].

One key feature of the original semantics is that they were not tied to any particular implementation of an event-based triplestore, removing the need to directly convert queries into corresponding triplestore queries. This was achieved by defining the semantic functions in terms of an abstract triplestore *interface*. The original semantic functions themselves were defined in pure math notation, suitable for implementation in any sufficiently powerful programming language.

In [10], a Miranda implementation of the semantics was demonstrated, and in [12] a Haskell implementation was produced as well. In both the Miranda and Haskell versions,

the sets used in the semantics were represented using singly-linked lists, called *lists*. Lists are composed using the `(:)` operator, where the left argument of `(:)` is an element and the right argument is a list. The `(:)` operator is *right-associative* prepends the left operand to the right operand. The empty list is denoted with `[]`. The syntax `[x_1, x_2, ... x_n]`, `(x_1 : x_2 : ... : x_n : [])` and `(x_1 : (x_2 : ... : (x_n : []))))` are equivalent. Lists are homogeneous in that types of all elements in the list must be identical.

Functions in Haskell that take two arguments may be treated as binary operators by surrounding the function name with backticks (‘) in a function call. For example, if the function `plus` takes two numeric arguments, then the Haskell expressions “`plus 1 2`” and “`1 ‘plus‘ 2`” are equivalent.

A summary of the original semantics and the Haskell implementation is provided below:

3.2.1 Triplestore interface

Triplestore access was implemented for in-program triplestores through the following functions, where `ev_data` is an in-program triplestore represented as a list of 3-tuples, each 3-tuple representing a triple:

```
getts_1 ("?",b,c)   = [x | (x,y,z) ← ev_data, y == b, z == c]
getts_2 (a,"?",c)   = [x | (x,y,z) ← ev_data, x == a, z == c]
getts_3 (a,b,"?")   = [z | (x,y,z) ← ev_data, x == a, y == b]
```

Other useful utility triplestore functions were defined in terms of the `getts_*` functions:

```
get_subj_for_ev ev       = getts_3 (ev,  "subject","?")
get_subjs_for_events evs = concatMap (get_subj_for_event evs)


get_members set = get_subjs_for_evs evs
where
  evs = intersect evs_of_type_membership evs_with_set_as_object
  evs_of_type_membership   = getts_1 ("?", "type", "membership")
```

```
  evs_with_set_as_object   = getts_1 ("?",  "object",  set)

get_subjs_of_event_type et  = get_subjs_for_evs evs
where
  evs = getts_1 ("?",  "type",  et)
```

3.2.2 Semantic functions

Common nouns were defined as "the set of entities that are members of the set associated with that noun"[12]. These were implemented using the `get_members` function.

```
person = get_members "person"
```

Intransitive verbs were defined as "the set of entities that are subjects of an event of the type associated with that verb"[12]. They were implemented using the `get_subjs_of_event_type` function.

```
steal = get_subjs_of_event_type "steal_ev"
```

Proper nouns were defined as "functions that take a set of entities as an argument and which return True if a particular entity is a member of that set"[12]. They were implemented using the `member` function which tests list membership.

```
torrio setofents  = "torrio" `member` setofents
```

Determiners were defined as functions taking two sets of entities, called a nounphrase and verbphrase respectively. Each of these functions is defined in terms of set intersection. They were implemented using list intersection:

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

Conjoiners for common nouns were implemented similarly:

```
nounand s t  = intersect s t
```

```
nounor  s t  = mkset (s ++ t) -- behaves like set union
that         = nounand
```

A *determiner phrase* was defined as being a determiner such as "a" or "every" applied to a common noun, for example "every moon".[12]. A *termphrase* was defined as being either a proper noun or a determiner phrase[12]. Conjoiners for termphrases were implemented as follows:

```
termand tmph1 tmph2 setofents = (tmph1 setofents) && (tmph2 setofevs)
termor  tmph1 tmph2 setofents = (tmph1 setofents) || (tmph2 setofevs)
```

This was necessary because common nouns were sets of entities, but proper nouns and determiner phrases were functions that acted on sets of entities.

Transitive verbs were defined in terms of *images*[12]. Briefly, images are constructed from binary relations with the `collect` function.

Definition 7 (Collect function)**.** *The function collect is defined such that it takes a binary relation as an argument, e.g. joinrel, and "returns a new binary relation, containing one binary tuple (x, image_x) for each member of the projection of the left-hand column of joinrel , where image_x is the mathematical image of x under the relation joinrel"[12]*

Definition 8 (Image (original semantics))**.** *A function that maps elements in the domain to sets*

Intuitively speaking, the `collect` function converts arbitrary binary relations into functions. $(\forall x)$ All pairs $(x,y_1),(x,y_2),...,(x,y_n)$ in the argument of `collect` are grouped into one pair $(x,\{y_1,y_2,...,y_n\})$ in the returned binary relation.

In the Haskell implementation, binary relations and images were represented using *association lists*, which are lists of pairs[12].

`collect` was implemented as follows:

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

According to [7], this implementation of `collect` has a worst-case asymptotic time complexity of $O(n^2)$.

Definition 9 (Entity-event relation)**.** *A binary relation from entities in a triplestore to events in that triplestore*

The function `make_image` creates an entity-event relation from a given event type and then converts it to an image using `collect`:

```
make_image et = collect
  [(subj, ev) | ev ← evs, subj ← getts_3 (ev, "subject","?") ]
        where evs = getts_1 ("?", "type",  et)
```

Transitive verbs were defined by *filtering* pairs in the image of the event type that corresponds to the verb using the termphrase provided. The only pairs remaining in the image are those for which the termphrase predicate returns `True`. An example is the "join" verb:

```
join tmph = [subj | (subj, evs) ← make_image "join_ev",
     tmph (concat [getts_3 (ev,  "object", "?") | ev ← evs])]
```

Prepositional phrases were defined through an extension of the above mechanism. Before passing an entity list to a termphrase, the events those entities were drawn from would be first filtered through a series of prepositions.

Prepositions were defined as a pair consisting of the name of a property of an event and a termphrase. Chained prepositional phrases were defined as a list of prepositions.

An example chained prepositional phrase: `[("with_implement", a telescope), ("year", year "1877")]`

`join` could be modified to support prepositions as follows:

```
join tmph preps = [subj | (subj, evs) ← make_image "join_ev",
  tmph (concat [getts_3 (ev,"object","?") | ev ← evs,
    filter_ev ev preps])]
```

where `filter_ev` is defined as follows[12]:

```
filter_ev ev [] = True
```

```
filter_ev ev (prep:list_of_preps)
  = ((snd (prep)) (getts_3 (ev,fst (prep),"?")))
      && filter_ev ev list_of_preps
```

3.2.3 Haskell implementation for SPARQL

In 2015, Agboola modified the Haskell implementation to support SPARQL endpoints, with some efficiency improvements[7]. In particular, the asymptotic time complexity of the `collect` function was improved to $O(n \log n)$ time.

3.3 Improvements over Original Semantics

In the following section, *function currying*[37] is used to simplify the description of the semantics. Briefly, a function f with n arguments can be *curried* into a chain of n functions, each accepting one argument and returning the next function in a chain, effectively "fixing" the current argument in the returned function. If the end of the chain is reached, the value that f would have produced had it been directly called with all n arguments is returned. This is a key feature of many functional programming languages[22].

We use the syntax $f\ x_1\ x_2\ \ldots\ x_n$ to denote a function or function call with n arguments. If this syntax is used and only a partial number of arguments are provided, then a function of the remaining arguments is returned. Otherwise, this syntax is equivalent to $f(x_1,x_2,\ldots,x_n)$.

We use set-builder notation to define the sets and binary relations throughout the semantic functions. When we provide a definition for a set R, we use the syntax:

$$R(a_1,a_2,...,a_n) = \{...\}$$

R's arguments (if any) are substituted into the set definition on the right hand side of the equation. We may combine this syntax with function currying, as used in the denotations for English words in our semantics, to simplify our definitions.-

The improvements from the original semantics presented in [11][10] are detailed below. This new semantics is called "Unified EV-FLMS", abbreviated as UEV-FLMS.

3.3.1 Multiple-property prepositions and terminology

Prepositions were previously defined as a pair consisting of the name of a property of an event and a termphrase. This meant that prepositions could only refer to one property of an event, making prepositions such as "in" impossible to express, as it could refer to either a location or a range in time. To solve this, we extended the definition of preposition in our semantics. First, some new terminology:

Definition 10 (Property)**.** *A* `predicate` *of a triple in an event-based triplestore.*

For example, in the triple `(event1000, with_implement, refractor_telescope_1)`, `with_implement` is a property. Since the triple belongs to `event1000`, we say that `with_implement` is a property of `event1000`. In general, there is no restriction on the number of entities of an event that share the same property.

Definition 11 (Preposition)**.** *A pair consisting of a set of properties and a predicate.*

By using a set of properties rather than a single property as in the original semantics, we are able to support prepositions that could refer to multiple properties. For example, the preposition `in'` in our semantics is defined as:

$$\texttt{in}'\ \texttt{tmph} = (\{\text{“}\texttt{location}\text{”}, \text{“}\texttt{year}\text{”}\}, \texttt{tmph})$$

When referring to the entities with property `prop` of an event, we are referring to the `objects` of the triples of the event that have the predicate `prop`. We may use the phrase "the `prop`s of an event" as a shorthand for "the entities with property `prop` of an event". For example, the `subjects`, `objects`, or `types` of an event are the entities with property `subject`, `object` or `type` of that event, respectively.

3.3.2 Naming and definition of "Images"

Originally, transitive verbs were defined in terms of *images*[12]. In this thesis report, we use the term *function defined by the relation* instead of *image*, as *image* is a term that already exists throughout mathematics and has a different meaning.

Definition 12 (Function defined by the relation *r*)**.** *The function defined by the binary relation r is the set* $\{(\texttt{x}, \texttt{image}_\texttt{x}) : \texttt{x}$ *is a member of the domain of* $\texttt{r}$ *and* $\texttt{image}_\texttt{x}$ *is the image of* $\texttt{x}$ *under* $\texttt{r}\}$

The function defined by a relation is referred to throughout this thesis report by the shorthand *FDBR*. It is represented by an association list in the Haskell code, as *images* were represented in the original semantics. The function defined by the relation *r* is denoted with the syntax $\texttt{FDBR}(\texttt{r})$. We call an element $(\texttt{x}, \texttt{image}_\texttt{x})$ of an FDBR an *FDBR-pair*.

We also define the `ENTEVPROP` relation. Informally, $\texttt{ENTEVPROP}(\texttt{evs}, \texttt{prop})$ is the entity-event relation in which the `props` of the events `evs` are related to the events that they belong to.

Definition 13 (`ENTEVPROP(evs, prop)` relation)**.**

$$\texttt{ENTEVPROP}(\texttt{evs}, \texttt{prop}) = \{(\texttt{ent}, \texttt{ev}) : \texttt{ent} \in \texttt{getts}(\texttt{ev}, \texttt{prop}, \texttt{ANY}) \ \wedge \ \texttt{ev} \in \texttt{evs}\}$$

The *ENTEVPROP_TYPE* relation is similar, accepting an event type as an argument instead of a set of events. *ENTEVPROP_TYPE*$(ev\_type, prop)$ obtains the set of events where the `types` of those events are the desired event type.

Definition 14 (`ENTEVPROP_TYPE(ev_type, prop)` relation)**.**

$$\texttt{ENTEVPROP\_TYPE}(\texttt{ev\_type}, \texttt{prop}) = \texttt{ENTEVPROP}(\texttt{evs}, \texttt{prop})$$
$$\texttt{where evs} = \texttt{getts}(\texttt{ANY}, \texttt{type}, \texttt{ev\_type})$$

In the original EV-FLMS semantics, the `make_trans` function can be defined using the above functions as follows:

```
make_trans event_type tmph =
        tmph(map fst FDBR(ENTEVPROP_TYPE(event_type,"subject")))
```

These relations form the basis on which our new semantics is defined.

3.3.3 The implicit 'and' problem and the problem of 'every'

In the original semantics there were two problems resulting from how prepositional phrases were implemented.

First, a query such as "the sun is orbited by every planet" would have returned `False` due to the way `filter_ev` was implemented.

This was because in `FDBR(orbit_)`, where `orbit_ = ENTEVPROP_TYPE(orbit_ev, "object")`, the FDBR-pair for `sun` would have the form: $(\texttt{sun}, [\texttt{event1000}, \texttt{event1001}, \ldots, \texttt{event1008}])$, where each event would denote a separate `orbit` event corresponding to each planet that orbits the sun. `filter_ev` applied each preposition's termphrase to each event separately, meaning that no information provided by other events could be used by the predicate. Hence `every planet` would have returned `False` for each individual event's associated `object`, and `filter_ev` would have discarded the pair for `sun` in $FDBR(\texttt{orbit\_})$. If there were one singular orbit event in which all of the planets were listed as orbiting the sun, then this query would have worked. However, if there are properties unique to the planets being described in each orbit event, then it is not possible to describe all orbits in one singular event. Therefore there was a need for a method to filter events in FDBRs that allowed termphrases to view the objects of *all* events in an FBDR-pair rather than just one.

Second, an implicit "and" was placed in between each preposition, transforming sentences such as "who discovered something with two telescopes in 1914" into "who discovered something with two telescopes and in 1914".

Although the above sentence is ambiguous in that it could be asking whether someone used two telescopes to discover one particular object, or whether someone discovered potentially different objects with two telescopes, the asker most certainly did not intend

for an implicit "and" to be inserted in between the prepositions. This happens because in the original `filter_ev` function, the sets of events were not actually "honed down" before being passed to subsequent predicates in the preposition chain.

In our semantics we define a new `filter_ev` function that overcomes both these problems. Its definition is deferred until the *Transitive Verbs* section as some additional concepts must be introduced first.

3.3.4 Semantic consistency

One goal of the new semantics is to unify concepts in the original semantics. A significant refactoring was performed in order to accomplish this task.

Triplestore utility functions

The `getts` functions are identical to how they were in[16], however all other triplestore utility functions have been modified from their original definitions. In particular, the functions `get_subjs_for_event`, `get_subjs_for_events`, `get_subjs_for_event_type`, and `get_members` return an FDBR rather than a set of entities. They are defined as follows:

$$\texttt{get\_subjs\_for\_events evs} = \texttt{FDBR}(\texttt{ENTEVPROP}(\texttt{evs}, \text{“}\texttt{subject}\text{”}))$$

$$\texttt{get\_subjs\_for\_event ev} = \texttt{get\_subjs\_for\_events}\{\texttt{ev}\}$$

$$\texttt{get\_subjs\_for\_event\_type ev\_type} = \texttt{FDBR}(\texttt{ENTEVPROP\_TYPE}(\texttt{ev\_type}, \text{“}\texttt{subject}\text{”}))$$

The definition of `get_members` is identical to the original except that it uses the new `get_subjs_for_events` function.

By returning FDBRs, information about the entire entity-event relation is provided rather than only information about entities belonging to those events. This change simplifies the definitions of the triplestore utility functions and also makes their function more uniform.

Semantic functions

All semantic functions have been modified to accept FDBRs and return FDBRs, with the exception of "query" functions `whatobj`, `where'`, `how'`, `when'`, and `what`.

`what` and `whatobj` are differentiated to obtain the `subjects` and `objects` of FDBRs, respectively, with the `objects` being obtained through the events in FDBR.

Determiner phrases and proper nouns Previously, predicates were functions from sets of entities to boolean values.

In our new semantics, we modified the definition of predicates such that predicates accept FDBRs and return FDBRs. A returned FDBR that is non-empty is considered to be `True`. A returned FDBR that is empty is considered to be `False`. Specifically, a predicate p evaluated on an FDBR F returns only the elements of F that are *relevant* to p.

Definition 15 (Relevant and irrelevant FDBR-pairs with respect to a predicate)**.** *An FDBR-Pair x of an FDBR F is* irrelevant *with respect to a predicate p if $p\ F = p\ F'$, where F' is obtained by removing x from F. Otherwise, x is* relevant *to p.*

Definition 16 (Predicate)**.** *A function from FDBRs to FDBRs that discards irrelevant FDBR-pairs from its argument.*

Most predicates in our semantics are defined in terms of the function `intersect_fdbr`:

Definition 17 (intersect_fdbr function)**.**

$$\texttt{intersect\_fdbr fdbr1 fdbr2} =$$

$$\{(\texttt{ent2}, \texttt{evs2}) : (\texttt{ent1}, \texttt{evs1}) \in \texttt{fdbr1} \wedge (\texttt{ent2}, \texttt{evs2}) \in \texttt{fdbr2} \wedge \texttt{ent1} = \texttt{ent2}\}$$

Briefly, this function performs the intersection of two FDBRs using the entity name in each FDBR-pair. The events from the second FDBR are preserved in the intersection, while the events from the first FDBR are discarded.

For an example of why `intersect_fdbr` is defined this way, consider the sentence `a moon spins`. The events that *justify* the claim that a particular moon spins are contained in the FDBR `spins`. On the other hand, the events of `moon` are only membership events and don't contribute any useful information about whether a given moon spins or not. Therefore

we would expect in the FDBR for `a moon spins`, we would only have events from the `spins` relation. Furthermore, it wouldn't make sense to have any entities that weren't moons in `a moon spins`, nor would it make sense for us to be missing any moons, since we know that all moons spin. If we let $\texttt{a} = \texttt{intersect\_fdbr}$, then we capture exactly these semantics.

In general, in any determiner function, we are interested in the events of the second FDBR as they provide *justification* that the entities in the first FDBR have some characteristic that the second FDBR expresses. The first FDBR is useful only for the entities contained within it as a way of specifying which entities should be queried. With this in mind, we define the determiners as follows:

$$\texttt{a} = \texttt{intersect\_fdbr}$$

$$\texttt{any} = \texttt{a}$$

$$\texttt{the} = \texttt{a}$$

$$\texttt{some} = \texttt{a}$$

$$\texttt{an} = \texttt{a}$$

$$\texttt{every} = \begin{cases} \texttt{intersect\_fdbr nph vbph, if nph\_entities} \subseteq \texttt{vbph\_entities} \\ \emptyset\texttt{, otherwise} \end{cases}$$

where

$$\texttt{nph\_entities} = \texttt{map fst nph}$$

$$\texttt{vbph\_entities} = \texttt{map fst vbph}$$

$$\texttt{one nph vbph} = \begin{cases} \texttt{intersect\_fdbr nph vbph, if } |\texttt{intersect\_fdbr nph vbph}| = 1 \\ \emptyset\texttt{, otherwise} \end{cases}$$

$$\texttt{two nph vbph} = \begin{cases} \texttt{intersect\_fdbr nph vbph, if } |\texttt{intersect\_fdbr nph vbph}| = 2 \\ \emptyset\texttt{, otherwise} \end{cases}$$

`fst` is a function that returns the first component in a pair. The `map` function computes the image of a function f over the elements of a set s. Hence the function `map fst` obtains all of the entities in an FDBR.

Proper nouns are defined as follows:

$$\texttt{make\_pnoun noun fdbr} = \texttt{intersect\_fdbr } \{(\texttt{noun}, \emptyset)\} \texttt{ fdbr}$$

Note that proper nouns are treated the same as determiners under the same reasoning.

This change was actually motivated by our changes to how prepositional phrases are handed in our semantics, and is the basis through which "filtering" occurs along preposition chains. Since predicates only return FDBR-pairs that are relevant to them, `filter_ev` can directly use predicates to filter FDBRs and find the events that are common among all of them, if any.

Transitive verbs In the original semantics, the function `make_trans` was used to construct an FDBR out of an event type.

In our semantics, we distinguish between the active and passive voices of transitive verbs. For example, in the sentence "Hall discovered a moon", the active voice is being used. In the sentence "A moon was discovered by Hall", the passive voice is being used. This voice of a verb changes what the verb is acting on in a sentence. To express this in the semantics, we provide two functions, `make_trans_active'` and `make_trans_passive'` to construct an FDBR denoting the active and passive voice of a transitive verb, respectfully.

$$\texttt{make\_trans\_active}' \ \texttt{ev\_type tmph preps} =$$

$$\texttt{filter} \left(\texttt{prepFilter} \left(\{(\{\text{“}\texttt{object}\text{”}\}, \texttt{tmph})\} \cup \texttt{preps} \right) \right) \left(\texttt{fdbr\_active} \right)$$

$$\texttt{make\_trans\_passive}' \ \texttt{ev\_type preps} =$$

$$\texttt{filter (prepFilter preps) (fdbr\_passive)}$$

where

$$\texttt{prepFilter}((\_, \texttt{evs})) = \texttt{filter\_ev preps evs}$$

$$\texttt{fdbr\_active} = \texttt{FDBR}(\texttt{ENTEVPROP\_TYPE}(\texttt{ev\_type}, \text{“}\texttt{subject}\text{”}))$$

$$\texttt{fdbr\_passive} = \texttt{FDBR}(\texttt{ENTEVPROP\_TYPE}(\texttt{ev\_type}, \text{“}\texttt{object}\text{”}))$$

The `filter` function takes a function that returns a boolean value and returns a new set containing only the elements of the original set for which that function returned `True`. `filter_ev` is used to filter FDBRs such that only FDBR-pairs that match the prepositions are kept. It is defined as follows:

$$\texttt{filter\_ev preps evs} = \begin{cases} \texttt{False, if filtered} = \emptyset \\ \texttt{True, otherwise} \end{cases}$$

where

$$\texttt{filtered} = \{\texttt{ev} : (\forall \texttt{propFDBR} \in \texttt{propertyFDBRs}) \ \texttt{ev} \in (\texttt{snd propFDBR})\}$$

$$\texttt{propertyFDBRs} = \left\{ \texttt{pred} \left(\bigcup_{\texttt{pName} \in \texttt{propNames}} \texttt{FDBR}(\texttt{ENTEVPROP}(\texttt{evs}, \texttt{pName})) \right) : (\texttt{propNames}, \texttt{pred}) \in \texttt{preps} \right\}$$

Informally, `filter_ev` computes an FDBR of the relevant properties for each preposition from all events, and evaluates the termphrases of each preposition on each corre-

sponding FDBR. The intersections of all events in all FDBRs are used as a way of "honing down" the events. If there are no events in common with any preposition, then the return value is `False`, otherwise at least one event satisfies all prepositions and the return value is `True`. This solves the problems with how prepositional phrases were handled in the original semantics.

We also provide two more functions, `make_trans_active` and `make_trans_passive`, transitive verbs without prepositional phrases:

$$\texttt{make\_trans\_active ev\_type tmph} = \texttt{make\_trans\_active}' \texttt{ ev\_type tmph } \emptyset$$

$$\texttt{make\_trans\_passive ev\_type} = \texttt{make\_trans\_passive}' \texttt{ ev\_type } \emptyset$$

The mechanism for filtering prepositional phrases is powerful enough that no extra work need be done for handling the termphrase associated with a transitive verb, for example `a moon` in `discover (a moon)`. By adding a *virtual preposition* to the set of prepositions, `({({"object"}, tmph)}`, `filter_ev` is able to subsume that functionality, simplifying the semantics.

Therefore, both voices of transitive verbs, and versions both with and without support for prepositional phrases, are handled uniformly and transparently.

As an example of how filtering using prepositional phrases works, consider the query "who discovered at mt_wilson in 1938". The FDBR of the verb "discovered" contains the FDBR-pair ("`nicholson`", {`event1056`, `event1057`, `event1058`, `event1059`}). In Figure 3.1, a visualization of the four events in the FDBR-pair with "nicholson" as the subject is shown. The matching properties queried by the prepositional phrase "at mt_wilson" and "in 1938" are highlighted in pink and purple, respectively. An FDBR is constructed for each prepositional phrase from the properties denoted in each preposition, and those FDBRs are filtered using each preposition's corresponding predicate. The set of events common to all filtered FDBRs for each prepositional phrase is contained within the dashed box. According to this chain of prepositions, the set of events that have properties matching all predicates in the chain is {`event1056`, `event1058`}. Since there exists a non-empty set of events relevant to all prepositional phrases in the chain, "nicholson" is included in the

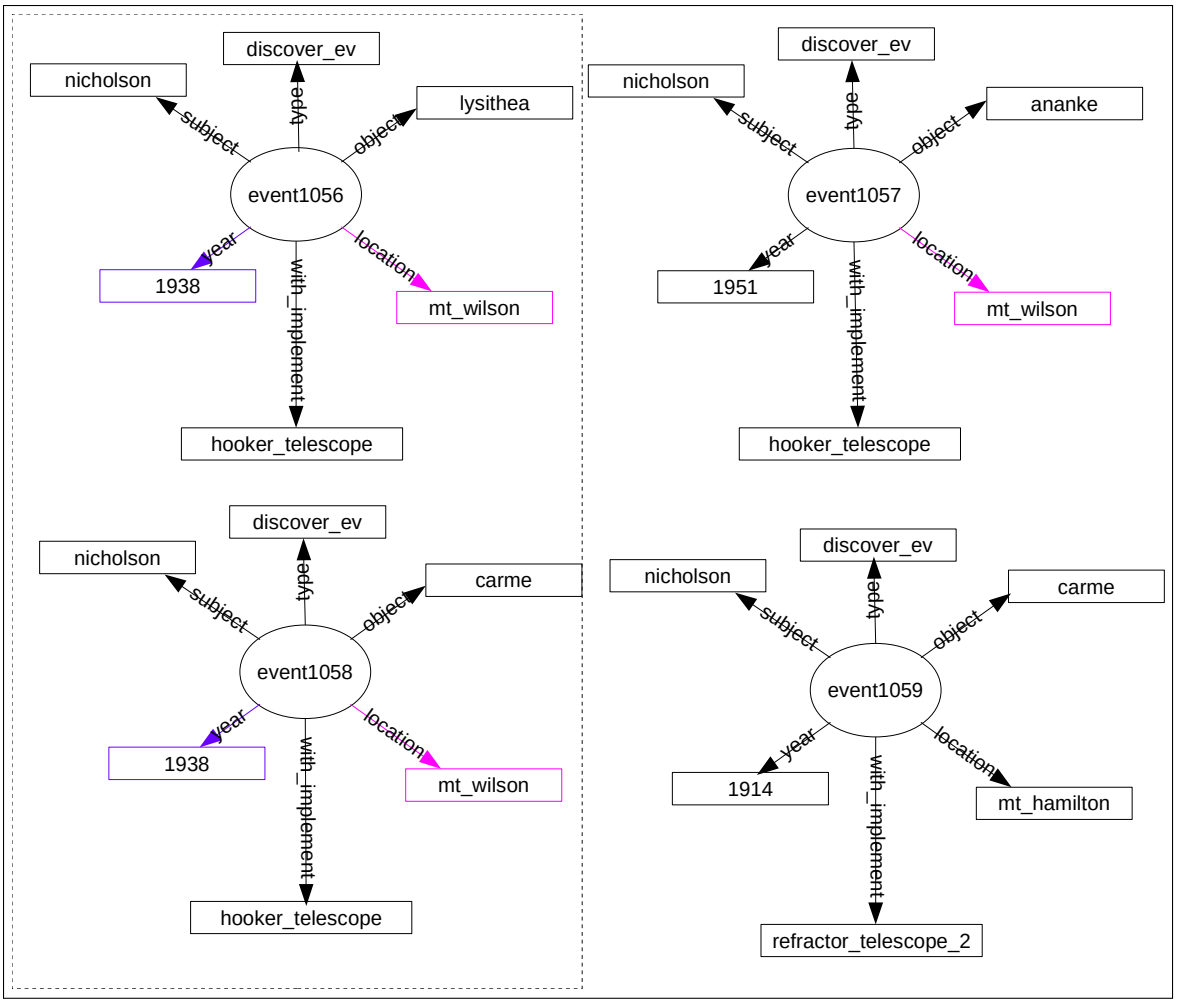


Figure 3.1: Example using prepositional phrases to filter events

query result. Next, consider the FDBR-pair with "hall" as the subject for the active "discover" verb: ("hall", {event1045, event1046}). Since both these events have property "year" as "1877", filtering using the prepositional phrase "in 1938" yields an empty set of matching events. Therefore, "hall" is not included in the query result.

Adjectives, common nouns, and intransitive verbs In the original semantics, intransitive verbs and common nouns were defined as a set of entities.

Since `get_subjs_of_event_type` and `get_members` return FDBRs, intransitive verbs and common nouns in our semantics are denoted by FDBRs. In addition, we also support adjectives in our semantics. Adjectives were not demonstrated in the original semantics,

however are easily accommodated using the `intersect_fdbr` function.

For example: "vacuumous moon", where `vacuumous` is an adjective and `moon` is a common noun, can be accomodated with `vacuumous ‘intersect_fdbr‘ moon`.

Conjoiners Originally, conjoiners for common nouns were defined in terms of the intersections and unions of sets of entities. In our semantics, conjoiners for common nouns are defined in terms of FDBRs.

$$\texttt{nounand} = \texttt{intersect\_fdbr}$$

$$\texttt{nounor nph1 nph2} = \texttt{nph1} \cup \texttt{nph2}$$

$$\texttt{that} = \texttt{nounand}$$

Similarly, the conjoiners for termphrases are also defined in terms of FDBRs:

$$\texttt{termand tmph1 tmph2 fdbr} = \begin{cases} \texttt{termor tmph1 tmph2 fdbr}, \texttt{ if } (\texttt{tmph1 fdbr} \cup \texttt{tmph2 fdbr}) \neq \emptyset \\ \emptyset, \texttt{ otherwise} \end{cases}$$

$$\texttt{termor tmph1 tmph2 fdbr} = \texttt{nounor } (\texttt{tmph1 fdbr}) \ (\texttt{tmph2 fdbr})$$

Note that `termand tmph1 tmph2` and `termor tmph1 tmph2` are predicates: they are functions from FDBRs to FDBRs and return exactly the FDBR-pairs that are relevant to them. `termand tmph1 tmph2` is curiously defined in terms of `termor`. The reason for this is that, provided both termphrases evaluate to `True` for FDBR F (i.e. the returned FDBRs are non-empty), all FDBR-pairs in both returned FDBRs are relevant: if any of those FDBR-pairs were to go missing from F, calling this new FDBR F', at least one of the termphrases `tmph1` or `tmph2` would evaluate to `False` on F', changing the result. Hence, all FDBR-pairs in the FDBRs returned by `tmph1` and `tmph2` when evaluated on F are relevant and should be included in the result of `termand tmph1 tmph2` when evaluated on F. On the other hand, if either termphrase evaluates to `False` on F, then `termand tmph1 tmph2`

evaluates to `False` on F, preserving the semantics of conjunction.

As an example, consider the semantic expression:

(`termand hall nicholson`) `discovered_intrans`

This expression corresponds to the sentence "`hall and nicholson discovered`". `discovered_intrans` is the intransitive form of the "discover" verb, which is an FDBR from the subjects of the events with type "`discover_ev`" to the set of events they are subjects of. `hall` and `nicholson` are predicates that filter FDBRs for the FDBR-pairs with"hall" and "nicholson" as a subject, respectively. Hence, filtering `discovered_intrans` with `hall` will yield an FDBR with one FDBR-pair:

{("`hall`", {`event1045`, `event1046`})}

Similarly, filtering `discovered_intrans` with `nicholson` will yield an FDBR with one FDBR-pair:

{("`nicholson`", {`event1056`, `event1057`, `event1058`, `event1059`})}

Since these FDBRs are both nonempty, both FDBRs are combined into one FDBR with `termand`:

{("`hall`", {`event1045`, `event1046`}),
("`nicholson`", {`event1056`, `event1057`, `event1058`, `event1059`})}

If either of these FDBR-pairs were removed from `discover_intrans`, then at least one of `hall` or `nicholson` would return an empty FDBR when evaluated on that new FDBR, changing the result. Therefore all FDBR-pairs in `hall discover_intrans` and `nicholson discover_intrans` are relevant and should be included in the result of `termand hall nicholson discover_intrans`.

3.3.5 The use of 'by' as a preposition

One item of note is that in our semantics we treat the word "by", as in "discovered by", as a preposition.

When designing the new semantics orginally, the termphrase after "by" was applied to a passive transitive verb in order to filter FDBR-pairs for those matching the termphrase. However, this is exactly the same task that `filter_ev` already performs when evaluating chains of prepositional phrases. By letting `by tmph = ({"subject"}, tmph)`, we can include `by` directly in chains of prepositional phrases as a *virtual preposition*, simplifying the grammar with no loss of functionality. With this approach, "by" can appear anywhere in a chain of prepositions, for example "in 1877 by hall", instead of needing to be directly next to the verb.

3.4 Summary

In this chapter we presented an overview of event-based denotational semantics. We first introduced the concept of an event-based triplestore, and then presented a summary of EV-FLMS, an event-based denotational semantics for use on those types of triplestores. We then presented a new event-based denotational semantics called Unified EV-FLMS or UEV-FLMS that improves on FLMS in several ways. First, we extended the definition of prepositions so that they may query multiple properties. Second, we chose a different name for the concept previously known as an "Image" in EV-FLMS that does not clash with existing nomenclature in mathematics. Third, we overcame two shortcomings with how prepositions were originally defined in EV-FLMS. Fourth, we unified the treatment of several distinct semantic concepts in EV-FLMS, simplifying the semantics without sacrificing any power. In the process of doing so, we discovered a novel way of handling the word "by", treating it directly as a preposition in our grammar. In Chapter 4, we describe the parser combinators used in our query program.

Chapter 4

Parser Combinators

To implement our Natural Language Interface, we integrated our semantics with a parser constructed as an executable attribute grammar. We chose to use the parser described by Frost and Hafiz in 2008[25] for this purpose, as it supports top-down parsing of ambiguous grammars. Our motivations for choosing this parser were threefold. First, it enables users to not have to worry about ambiguity or left-recursion in their grammars. The parser itself tracks ambiguity and evaluates all unique possible parses[25]. Second, both semantic and syntactic rules can be defined together, improving modularity[25]. Third, new syntactic and semantic rules can be easily and naturally coded in an attribute grammar that supports left recursion, thereby improving extensibility[25].

We could not use the parser as-is, however, since it did not support non-referentially transparent functions as attributes. We detail the modifications we made to the original parser in order to lift this restriction in Section 4.1 and in Section 5.4.

4.1 Handling non-referentially transparent functions

The parser combinators as described in this Thesis differ from their original implementations as described by Frost and Hafiz[25]. The most significant change that we made is that the combinators are now *monadic* in nature rather than being strictly pure functions. Briefly, monads in the Haskell programming language are types that are instances of the *Monad* typeclass that obey the *monad laws*[35]. In Haskell, functions that are not *referen-*

tially transparent are represented using computations in the monad `IO`, commonly referred to as the *IO monad*.

By modifying the parser to work with monadic rather than pure values, the restriction that the semantics themselves must also be pure was lifted, and we can safely support streaming information from external triplestores in our semantics as a result.

The parser works in the IO monad currently, however if other monads were desired, only minimal changes would be required to the parser in order to accommodate other instances of the Monad typeclass. In Chapter 9, one potential application of this functionality is discussed.

4.2 Summary of the Parser Combinators

Aside from the differences noted above, the combinators function the same as they did originally in [25]. They are summarized as follows:

- `(<|>)` – a combinator that represents an alternative. In the expression "`a <|> b`", both `a` and `b` are attempted to be matched against the string. If both `a` and `b` match, i.e. the grammar is ambiguous, both parse trees are returned in the result.

- `(*>)` – a combinator that represents a sequence. In the expression "`a *> b`", the parser would try to match `a` followed by `b`. Both must be matched in order for the parse to succeed.

In Chapter 5, we show how we constructed a parser as an executable attribute grammar using these combinators and integrated it with our novel event-based denotational semantics to produce a Natural Language Interface to the Semantic Web.

Chapter 5

The Query Program

5.1 Implementation language

We chose to implement UEV-FLMS in the Haskell programming language[22]. Briefly, Haskell is a lazily-evaluated functional programming language with first class support for monads. We summarize these concepts and provide justification for our choice in the following subsections:

5.1.1 Functional Programming

Functional programming languages are declarative, rather than imperative, in nature. This means that programs are written by composing functions together, much like how mathematical functions can be defined in terms of function composition. Unlike imperative languages, mutable state is avoided in programs written in functional languages. This lends lends itself to a highly expressive style of programming that heavily encourages code reuse while still retaining the ability to easily reason about code.

5.1.2 Lazy-evaluation

In a language that is *lazily evaluated*, expressions and subexpressions are evaluated only as they are needed at run-time. This enables infinitely recursive data structures to be expressed directly in the language, and allows for improved performance in some cases by leaving

unnecessary calculations unevaluated.

For example, consider the function `primes` that returns a list of all prime numbers. If no elements of `primes` are used, then no computation is performed at all. If only the first 20 elements of `primes` are used, only the first 20 elements need to be evaluated, with the rest being left unevaluated. In an eagerly evaluated language, every element of the list returned by `primes` would be computed the first time it was referenced, resulting in a non-terminating loop. In this example, lazy-evaluation enables users to use an intuitive abstraction to iterate through a sequence of prime numbers while avoiding the costs associated with those abstractions in eagerly evaluated languages.

5.1.3 Monads

Monads in functional programming are inspired from their counterparts in category theory. In a functional programming language, a type A is a monad if there exists definitions of two functions in that language with the types of those functions as follows:

- `bind` : $A\ x \rightarrow (\lambda x \rightarrow A\ y) \rightarrow A\ y$
- `return` : $x \rightarrow A\ x$

The `bind` function is commonly used as a binary operator. In Haskell, the bind function is denoted with operator $>\!\!>\!\!=$. In addition, these definitions must satisfy the *monad laws*. We use this syntax to summarize the monad laws as follows. Here, $a \equiv b$ denotes that the expression a is semantically equivalent to the expression b and hence they are interchangeable. These laws are defined using lambda calculus.

Monad Laws

$(\forall x)(\forall y)(\forall f)(\forall m)(\forall a)\ f$ is a function with type $f : x \rightarrow A\ y$, m is a value with type $m : A\ x$, and a is a value with type $a : x$

- Left-identity: `return` $a\ >\!\!>\!\!=\ f\ \equiv\ f\ a$
- Right-identity: $m\ >\!\!>\!\!=$ `return` $\equiv\ m$

- Associativity: $(m \gg\!= f) \gg\!= g \equiv m \gg\!= (\lambda x \rightarrow f\ x \gg\!= g)$

In Haskell, computations with monads are expressible with a special syntax called the "do" syntax. This allows for a convenient way of expressing monadic computations directly within the language. Haskell gives special treatment to the monad `IO`, enabling computations within it to be non-referentially transparent. An example expression in Haskell using the IO monad is as follows:

```
main :: IO ()
main = do
        line ← readLine
        putStrLn ("Hello world! " ++ line)
```

This syntax is equivalent to the code:

```
main :: IO ()
main = readLine >>= (λline → putStrLn ("Hello world! " ++ line))
```

Monads are useful to express computations that involve multiple independent actions with an implicit action that "ties" them together. This can be used, for example, to implicitly pass state as an argument through functions, a technique used in Frost and Hafiz's parser to efficiently parse highly ambiguous left recursive grammars[25].

5.1.4 Why Haskell?

In particular, mathematical functions are easily implemented in functional languages. Due to their declarative nature, often times the definition of the mathematical function itself can be directly expressed in the language. Since UEV-FLMS is heavily rooted in set-relational theory[39], we were able to implement all of our semantic functions as they were defined in Chapter 3 by directly expressing those functions in Haskell. To accommodate communication with external triplestores, we "lifted" the implementations of the semantic functions into the IO monad, enabling our semantics to use non-referentially transparent `getts` functions. We were also able to modify the original parser that Frost and Hafiz described[25] to support non-referentially transparent attributes and integrate that with our semantics, owing to the reusability of code that functional languages encourage.

It certainly would have been possible to use an imperative language to achieve the same end-result, however significantly more work would have been involved in doing so. For one, implementation of UEV-FLMS semantics in an imperative language would have taken much more code, as the mathematical definitions themselves would not be directly expressible in the language. Another difficulty would have been in parsing English queries, since adapting an existing parser for this end may not have been possible. Certainly, one would not be able to directly express an Executable Attribute Grammar directly in an imperative language without significant work, meaning that the semantics and the parser would not be able to be defined together in a piecewise fashion.

5.2 Data representation

Like the original Haskell implementations, we represent sets in our implementation using lists and binary relations as association lists. We represent triples using 3-tuples and we represent URIs using the `String` type, which is a list of `Char`.

5.3 Structure

A graph representing the structure of the modules in the XSaiga package in relation to one another is presented in Figure 5.1. Briefly, there exists an arrow from module `A` to module `B` in the graph if and only if module `A` imports module `B` in the source code.

5.4 AGParser2 and TypeAg2 modules

The parser is defined in the `AGParser2` module.

The types of the semantic functions in `SolarmanTriplestore` are defined in the TypeAg2 module. These are the implementations of the semantic functions described in Chapter 3. The TypeAg2 module therefore serves as the interface between the parser and the semantics.

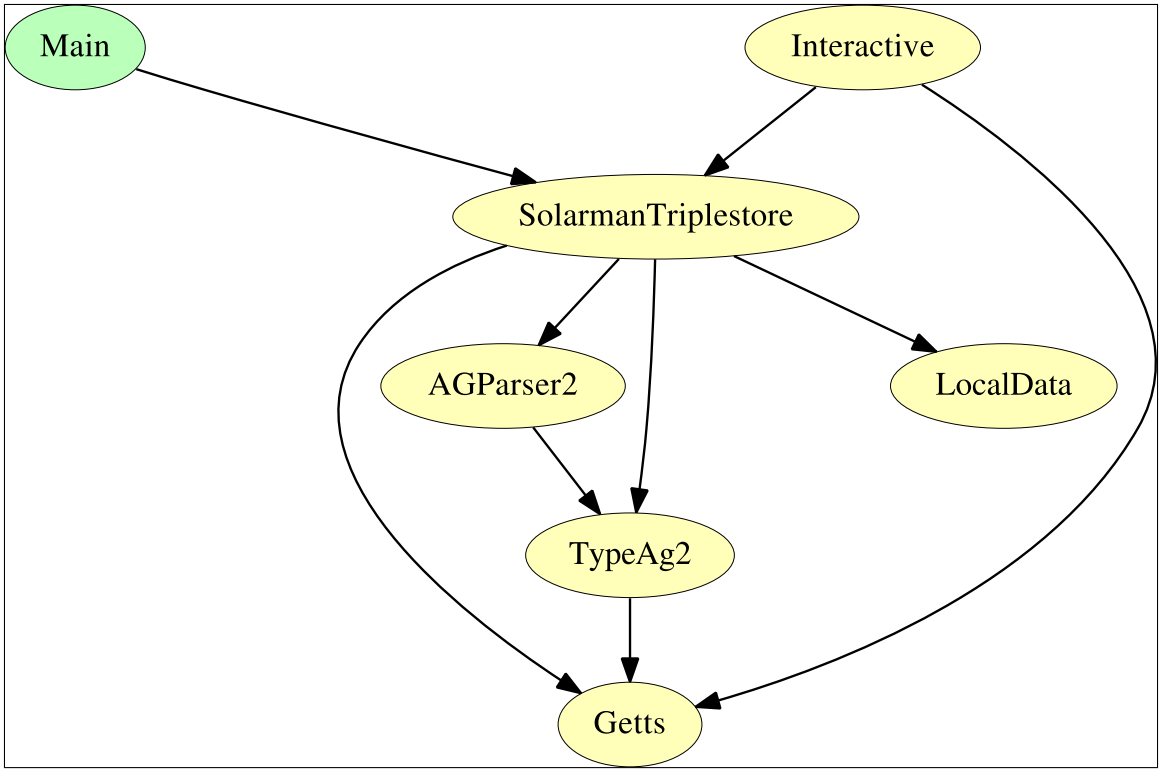


Figure 5.1: Module graph of the XSaiga package

The changes made to the original parser to accommodate monadic code are summarized below:

5.4.1 PrettyPrinting

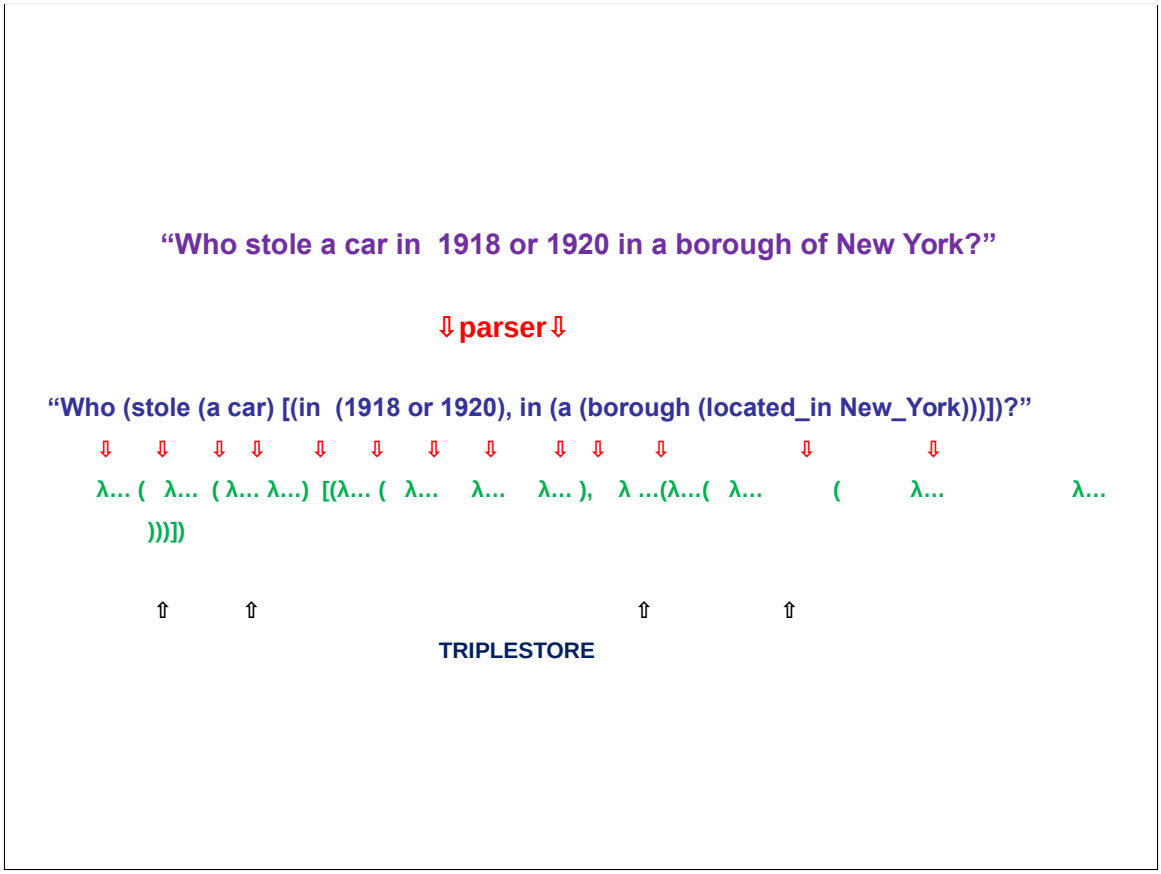
Utility functions for formatting the parser results had to be modified.

Pure version

In the pure version, we have:

```
class PP' a where
pp' :: a → Doc
```

where `Doc` is a formatted `String` and instances are defined for the various types used by the parser itself.



URL: http://cs.uwindsor.ca/~richard/semantics_presentations/talk_for_GraphQ.ppt

Figure 5.2: Parser operation: how an English sentence is mapped to semantic functions for evaluation[11]

Monadic version

In the monadic version, we have:

```
class PP' a where
pp' :: a → IO Doc
```

Here, the `pp'` function returns an IO action rather than a pure value. All instances of `PP'` were modified accordingly.

5.4.2 formatAttsFromAlt

Pure version

```
formatAttsFinalAlt  key e t  =
```

```
[(pp' [vcat [(vcat [vcat [vcat [text (show ty1v1)  |ty1v1←val1]
|(((st,inAtt2),(end,synAtts)), ts)←rs, end == e]
| ((i,inAt1),((cs,ct),rs)) ← sr ])
| (s,sr) ← t, s == key ]
```

Monadic version

```
formatAttsFinalAlt  key e t  =
sequence [(sequence [liftM vcat $ sequence
[(liftM vcat $ sequence
[liftM vcat $ sequence
[liftM vcat $ sequence
[liftM text (showio ty1v1) | ty1v1←val1]
|(id1,val1)←synAtts]] )
|(((st,inAtt2),(end,synAtts)), ts)←rs, end == e]
| ((i,inAt1),((cs,ct),rs)) ← sr ]>>=pp'
| (s,sr) ← t, s == key ]
```

5.5 Main module

The `Main` module implements a CGI interface for evaluating Natural Language queries using the `SolarmanTriplestore` module.

5.6 Interactive module

The `Interactive` module is used by the Direct Query Interface to directly evaluate semantic functions. It is intended to be used with SafeHaskell in order to restrict the evaluation of functions to a trusted subset, suitable for online interfaces.

In `SolarmanTriplestore`, a dictionary is defined that maps words to semantic functions. This module defines variables that are named after those words such that those functions can be directly accessed in a Haskell interpreter. This enables, for instance,

```
hall $ discovered phobos
```

to be directly evaluated at a Haskell prompt.

A Haskell file InteractiveGenerator.hs is used to generate this module using the dictionary in `SolarmanTriplestore`.

5.7 LocalData module

This module contains an in-program version of the triplestore located on our SPARQL endpoint. As the `Getts` module provides a general interface to triplestores in the form of a typeclass, we are able to support both in-program triplestores as well as remote triplestores. The module exports the list of triples as the variable `localData`:

```
localData = [("event1000", "object", "sol"),
("event1000", "subject", "mercury"),
("event1000", "type", "orbit_ev"),
("event1001", "object", "sol"),
("event1001", "subject", "venus"),
("event1001", "type", "orbit_ev"),
("event1002", "object", "sol"),
("event1002", "subject", "earth"),
("event1002", "type", "orbit_ev"),
("event1003", "object", "sol"),
... ]
```

5.8 SolarmanTriplestore and Getts modules

The implementation of our semantics in Haskell is contained within these modules, along with the parser constructed as an executable attribute grammmar and the dictionary used by the parser. We detail our implementation improvements over EV-FLMS in Section 5.9.

5.9 Improvements over Original Haskell Implementations

5.9.1 Type safety

The original semantics were implemented as *pure functions* in Haskell, which was acceptable for the in-program triplestores they were used on.

In [7], the `getts_*` functions were modified to retrieve triples from external SPARQL endpoints, enabling the original semantics to work with SPARQL triplestores. For SPARQL triplestores, however, the `getts_*` functions as defined in [7] are not actually guaranteed to be referentially transparent. In particular, SPARQL triplestores are able to change over time with triples being potentially added, removed, or modified. For example, consider the query "which people discovered a moon that was discovered by a person". "people" and "person" are synonyms in our semantics and therefore the same query would be performed twice. If the SPARQL triplestore changed in between these evaluations, then these queries could return different results, violating referential transparency.

The function `unsafeDupablePerformIO` was used in [7] to force Haskell to treat the `getts_*` functions as pure functions in order to maintain compatibility with the original semantics. The problem with `unsafeDupablePerformIO` is that it subverts the type system of Haskell. Code that is built using it is therefore not on solid theoretical ground within the constructs of the language, and surprising effects can occur as a result. The use of `unsafeDupablePerformIO`, while legitimate in some cases, is heavily discouraged within the Haskell community[5].

In this Thesis, we chose a different approach to handling external triplestore queries by representing the triplestore functions and semantics in terms of *monadic functions*. By expressing the semantics and triplestore functions monadically, we stay safely within the confines of Haskell's type system, avoiding the need to use `unsafeDupablePerformIO` in order to perform queries to external triplestores as a result. Another key benefit of this approach is that it preserves the compositional nature of the original semantics.

The semantics are implemented in the IO monad currently. However, if other monads were desired, just as with the parser described in Chapter 4, only minimal changes would be required in order to accommodate other instances of the Monad typeclass. In Chapter 9,

one potential application of this functionality is discussed.

5.9.2 The Getts module: A generic interface to triplestores using a typeclass

Typeclasses are used in Haskell to enable *ad-hoc polymorphism* in the definition of functions in the language. This can be used to provide generic interfaces to different types, without callers needing to be aware of the differences between those types. We used this feature of the language to provide a generic interface for triplestores in the form of typeclass `TripleStore`.

`TripleStore m` subsumes the functionality that the `getts_*` functions provided in the original semantics.

```
class TripleStore m where
getts_1 :: m → (Event, String, String) → IO [String]
getts_2 :: m → (Event, String, String) → IO [String]
getts_3 :: m → (Event, String, String) → IO [String]

getts_fdbr_entevprop_type :: m → String → String → IO FDBR
getts_fdbr_entevprop_type ev_data ev_type entity_type = do
evs ← getts_1 ev_data ("?", "type", ev_type)
getts_fdbr_entevprop ev_data entity_type evs

getts_fdbr_entevprop :: m → String → [Event] → IO FDBR
getts_fdbr_entevprop ev_data entity_type evs = do
pairs ← liftM concat $ mapM (λev → do
ents ← getts_3 ev_data (ev, entity_type,"?")
return $ zip ents (repeat ev)) evs
return $ collect pairs

getts_members :: m → String → IO FDBR
```

```
getts_members ev_data set = do
evs_with_set_as_object ← getts_1 ev_data ("?", "object", set)
evs_with_type_membership ← getts_1 ev_data
("?", "type", "membership")
getts_fdbr_entevprop ev_data "subject" $
intersect evs_with_set_as_object evs_with_type_membership
```

First and foremost, the `getts_*` functions defined in `TripleStore m` now properly return *IO actions*. Three new functions are introduced: `getts_fdbr_entevprop`, `getts_fdbr_entevprop_type`, and `getts_members`. `getts_fdbr_entevprop_type` serves the same purpose that `make_image` had in the original semantics. These functions are named after their counterparts in Chapter 3.

Only the three `getts_*` functions must be defined for the new type of triplestore at minimum. However efficient implementations of all functions in the typeclass may be provided if desired. We provide a backend using a SPARQL endpoint as a triplestore using the `SPARQLBackend` type and a backend for in-program triplestores using the `[Triple]` type as instances of `TripleStore`.

Basic query fusion

In addition to this, a basic form of query fusion has been implemented in the form of memoization. Briefly, queries and their results are stored in *key-value stores*. When a query is performed, it is first checked against the appropriate key-value store to see if the same triplestore query has been made previously. If it has, the previous result is returned. Otherwise, the request is made to the remote triplestore and its result is saved into the appropriate key-value store. Multiple requests for the same information to the remote triplestore are therefore fused together.

The key-value stores are held in top-level mutable variables. Defining top-level mutable variables in Haskell is a subject that has been explored in depth, with many proposals in how to provide an idiomatic method in the language to express it. According to the Haskell community, the accepted way of doing this for now is with the following pattern[5]:

- A top-level mutable variable `v` is defined using `v = unsafePerformIO $ newIORef value`
- `v` must be annotated with the compiler pragma {`-# NOINLINE v #-`}

Efficiency of "collect"

The `collect` function as defined in [7] used a key-value store from the `Data.Map.Lazy` module in Haskell to efficiently construct Images from relations represented as association lists. Because all key-value pairs of the `Map` will be traversed in order, immediately and in all cases, in this thesis the `Data.Map.Strict` module was used instead. The asymptotic time complexity of both methods are identical, however the `Strict` version uses less memory and is slightly faster as it does not attempt to store partially evaluated areas of the Map as *thunks*, which are partially evaluated Haskell expressions.

The `collect` function is defined in this module as:

```
collect = Map.toList ∘ Map.fromListWith (++) ∘ map (λ(x, y) → (x, [y]))
```

Efficiently constructing FDBRs from grouped association lists

In addition, a new function, `condense`, has been created to efficiently construct $FDBR(r)$ such that r is represented by a grouped association list.

Definition 18 (Grouped Association List)**.** *An association list where the indices of all pairs in the list with equal first components are contiguous An association list where all pairs with equal first components are contiguous in the list*

We refer to any association list that is not a grouped association list as an *ungrouped association list*.

The `condense` function is defined in this module as:

```
condense =
map (λlist → (fst $ head list, map snd list)) ∘ List.groupBy cmp
where
cmp x y = (fst x) == (fst y)
```

In our SPARQL backend, the `getts_fdbr_entevprop` and `getts_fdbr_entevprop_type` functions efficiently construct FDBRs from their ENTEVPROP relations using this function by requesting to the SPARQL endpoint that the ENTEVPROP query results be sorted according to the first element in each pair. Since this groups pairs together in the association list by their first element, `condense` is used instead of `collect` in order to construct the FDBR in the SPARQL backend.

The `condense` algorithm can be expressed in functional language pseudocode as follows. In this pseudocode, we denote lists with the syntax $[a_1, a_2, \ldots, a_n]$, with the empty list denoted as []. Lists are represented as a recursive abstract datatype using the `cons` function as originally used in lambda calculus to represent lists using Church Encoding[17]. The syntax $[a_1, a_2, \ldots, a_n]$ is hence equivalent to the expression:

$$\texttt{cons}\ a_1\ (\texttt{cons}\ a_2\ (\ldots(\texttt{cons}\ a_n\ [])))$$

We denote tuples with the syntax $(a_1, a_2, \ldots, a_n)$. Function application is denoted as in Chapter 3. For example: `head [x, y, z] => x` and `tail [x, y, z] => [y, z]`. Function definition is denoted with a similar syntax, supporting *pattern matching* on abstract datatypes such as tuples and lists. For example, the function `f (cons x xs) = x` matches its first argument with the outer `cons` in a list, and the function `g (x, y) = x` matches its first argument with a pair, assigning variables `x` and `y` to the first and second components of that pair, respectively. A `where` clause may be appended to a function definition to define variables used within that function. A let binding, using the form `let pattern = expr in expr'`, creates a local variable binding using pattern matching in an expression. The expression `expr` is assigned to the pattern `pattern`, whose variables are made available in `expr'`. Finally, conditional evaluation is expressed with the `if expr then expr1 else expr2` expression: if `expr` evaluates to `True`, then the result of the expression is `expr1`, otherwise it is `expr2`.

These syntax choices were chosen to both simplify expression of the algorithm and ease implementation of the algorithm in a wide variety of programming languages.

```
ALGORITHM condense
INPUT: al (grouped association list)
OUTPUT: fdbr (FDBR of al as an association list)

condense alist = map mkpair (groupBy cmp alist)
mkpair list = (head list, map snd list)

fst (a, b) = a
snd (a, b) = b

head (cons x xs) = x
tail (cons x xs) = xs

cmp x y = (fst x) == (fst y)

map f [] = []
map f (cons x xs) = cons (f x) (map f xs)

span p [] =  ([], [])
span p (cons x xs')
 = if (p x)
   then let (ys,zs) = span p xs' in (cons x ys,zs)
   else ([],(cons x xs'))

groupBy eq []          = []
groupBy eq (cons x xs) = cons (cons x ys) (groupBy eq zs)
where (ys,zs) = span (eq x) xs

fdbr = condense al
```

Theorem 5.9.1. *Algorithm* `condense` *has* $O(n)$ *worst-case time complexity, with comparison on the first element in the association list pairs being the key operation.*

Proof:

`cmp` is the function that compares two association list pairs by their first element.

Let `al` be a grouped association list.

`condense al => map mkpair (groupBy cmp al).`

Proposition 5.9.1. *The function* `groupBy cmp` *has* $O(n)$ *worst-case time complexity, with comparisons on the elements of the input list using* `cmp` *being the key operation.*

Proof:

Let `lst = (cons x xs)` be a list with n elements.

In `groupBy cmp (cons x xs)`, argument `eq = cmp` and the predicate used in `span` is `p = eq x = cmp x`.

The function `span` returns a partition `(ys, zs)` of the input list `xs`, where `ys` is the longest prefix of `xs` such that predicate `p` is `True` on all elements in the prefix, and `zs` are the remaining elements in `xs`. It follows that `p` is evaluated at least s times and at most $s+1$ times, where s is the length of `pre` (it will only be evaluated s times if `pre = xs`).

By recursing into the second list returned by `span`, no previous elements of `lst` are revisited, and no elements are skipped, so `groupBy` partitions the input list `lst` into m lists. Call this partition `part`. Note that the sum of the lengths of all lists in `part` is n.

For each list `i` in `part` except the last, `((length i) - 1) + 1 = length i` comparisons will have been made (`groupBy` calls span on `xs`, not `cons x xs`). For the last list *last* in `part`, `(length last - 1)` comparisons will have been made (the longest prefix is `xs` in this case). Therefore, the total number of comparisons made can be expressed by:

```
length part[0] + length part[1] + ... + length part[m - 2]
  + (length part[m - 1] - 1)
= (length part[0] + length part[1] + ... + length part[m - 1]) - 1
= n - 1
```

Hence, the worst-case time complexity of `groupBy cmp` is $O(n-1) = O(n)$, with comparisons using `cmp` being the key operation. □

–

`mkpair` performs no comparisons and therefore has a worst-case time complexity of $\theta(1)$ with comparisons using `cmp` being the key operation.

–

The `map` function evaluates `mkpair` over every list in the partition returned by (groupBy cmp al). Since neither `map` nor `mkpair` perform any comparisons using `cmp`, they do not contribute to the number of key operations performed.

Therefore, if the length of `al` is n, the worst-case time complexity of `condense` is

$$O(1) + n * (O(1) + O(1)) + O(n) = O(n)$$

with comparison on the first element in the association list pairs (`cmp`) being the key operation. ■

5.10 Summary

In this chapter we presented an overview of our query program structure as well as how we efficiently implemented our semantics in Haskell. We showed the modifications we made to the parser described in [25] in order to accommodate monadic functions as attributes. We presented a basic form of query fusion as used by our implementation, and showed an improved version of `collect` for grouped association lists. In Chapter 6, we perform some benchmarks to measure the empirical performance of our code.

Chapter 6

Timing

6.1 Experiment setup

We conducted some experiments in order to measure the performance of our implementation. First, we compiled all Haskell code with profiling enabled such that we were able to accurately see how much time was spent in various functions. Second, we enabled the highest level of optimization possible in the GHC Haskell compiler as we wanted to measure how performance would look in the real world.

This was accomplished by using the following arguments to GHC when compiling:

```
ghc -prof -fprof-auto -rtsopts -O3
```

Note that to replicate these experiments, you must rebuild the entire package library that ships with GHC with profiling enabled. This is because with GHC, code that is instrumented with profiling must only link to other code that is instrumented for profiling.

All tests were performed on a system with these specifications:

- Intel Core i7 4770k processor
- 16 GB of RAM
- Samsung 850 EVO Solid State Drive

6.2 Experiment description

In our first experiment, we do a simple measurement to see the amount of CPU time is spent for constructing the FDBR of binary relations of various sizes (recall that binary relations and FDBRs are represented with association lists in our code, detailed in Chapter 5).

In our second experiment, we examine the amount of time it takes to perform a query, using profiling information to break down exactly where most time is spent in processing.

6.3 Experiment 1

We construct both a grouped association list and an ungrouped association list with 10,000 unique events and 1000 entities with a varying number of pairs in both association lists. Definitions can be found in Section 5.9.2 for grouped association lists and ungrouped association lists. We chose to use fewer entities than events to ensure that the same entities were reused throughout different events, as would be seen in actual triplestores. We detail the construction of these association lists as follows.

Random number generation We use a uniformly distributed pseudorandom number generator to generate event identifiers and entity identifiers, representing these identifiers as strings. We use the seed 1024 when initializing the pseudorandom number generator in all cases, in order to make results more easily comparable between the implementations.

Ungrouped association lists For each ungrouped association list of size n, we use a uniformly distributed pseudorandom number generator to generate n event identifiers in the range of $[1, 10000]$ and n entity identifiers in the range of $[1, 1000]$, treating the result as a string. For entities, the word "entity" is prepended to the identifier. For example, if we generate 9900 as an identifier for an event, the identifier will be a string with name "event9900". We then directly combine both lists of identifiers pairwise to form an association list. For example, $n = 2$ and we generated entity names "5" and "100" and event names "event9" and "event500", these would be combined to form the association list $[(\text{“5”}, \text{“}\mathit{event}9\text{”}), (\text{“100”}, \text{“}\mathit{event}500\text{”})]$.

Grouped association lists For each grouped association list of size n, we used a uniformly distributed pseudorandom number generator to generate n event identifiers just as we did for ungrouped association lists, however we generate the list of entity identifiers differently. To generate the list of entity identifiers, we calculate $m = n/1000$ and for each i in the range $[1, 1000]$, we repeat i represented as a string m times in a list to obtain a list of n entities. We chose values of n such that 1000 divides n to ensure that each entity occurs exactly m times. When combining the list of entity identifiers and the list of event identifiers pairwise, the result is a grouped association list.

We compare the implementation of our `collect` and `condense` functions versus the previous implementation in [7] by evaluating them on the constructed association lists. Definitions for both implementations of `collect` and `condense` can be found in Section 5.9.2. For ungrouped association lists, we only compare the `collect` functions since `condense` may only be used with grouped association lists.

6.3.1 Results

Ungrouped association lists

First, we compare the implementations on ungrouped relations.

Pairs	Previous `collect`	Our `collect`
100,000	0.130 sec	0.129 sec
1,000,000	1.426 sec	1.388 sec
10,000,000	14.274 sec	14.283 sec

Grouped association lists

Next, we compare the implementations for grouped relations, including the `condense` as defined in Section 5.9.2.

Pairs	Previous `collect`	Our `collect`	`condense`
100,000	0.083 sec	0.068 sec	0.030 sec
1,000,000	0.775 sec	0.756 sec	0.401 sec
10,000,000	8.169 sec	7.618 sec	4.502 sec

6.3.2 Discussion

In the ungrouped case, both implementations are highly comparable, with no noticeable difference in running time between them.

In the grouped case, both `collect` implementations fare much better with increasing n. The reason for this is that both implementations use the Map datatype in Haskell, which is a key-value store that internally is represented as a balanced binary tree. Since the input grouped association list is already sorted on the entities in the association list, this makes building a balanced binary tree out of the elements a simple task. Both `collect` implementations are again highly comparable. The `condense` function demonstrates a clear improvement with larger values of n over the two `collect` implementations, as expected from the complexity analysis performed in Section 5.9.2.

6.4 Experiment 2

We performed two Natural Language queries to our SPARQL endpoints using a simple command line interface for Solarman:

1. "which vacuumous moon that orbits jupiter was discovered by nicholson or hall with a telescope in 1938 in mt_wilson or mt_hopkins"

2. "what was discovered in 1877 at us_naval_observatory"

We ran our command line interface with the following arguments to enable profiling for IO:

```
./solarman_cmd <query_string> +RTS -pa
```

We also ran our command line interface with these arguments to enable profiling for CPU time:

```
./solarman_cmd <query_string> +RTS -p
```

After doing so, we examined the "prof" files produced by each run of our command line interface.

When profiling for IO, we were interested in how much time was spent in the "SYSTEM" module, which is the interface that code in Haskell uses to communicate with the underlying operating system. Any time spent waiting on operating system interrupts is represented as time spent in the SYSTEM module.

When profiling for CPU time, the time spent in the SYSTEM module is excluded from the "prof" files. Since any IO request necessarily involves some amount of CPU overhead, we have included the CPU overhead involved in performing any IO request in these results. A brief explanation is given below for the four categories used:

- SPARQL query generation and result processing including XML parsing: The time spent in the Data.RDF module provided by the HSparql Haskell package. This includes the time spent generating SPARQL queries and parsing the results of those queries as represented in XML. This excludes the underlying network related processing detailed below.

- Network related processing: The time spent in the Network module, used for structuring requests over different protocols to be sent over networks. This excludes any time actually spent waiting on IO, and only measures the CPU overhead involved in generating these requests.

- Natural Language parsing: The time spent in the AGParser2 module (described in Chapter 4), excluding the time spent in the semantic functions, detailed below.

- Semantic functions: The time spent spent in the semantic functions in the SolarmanTriplestore module (described in Chapter 5), excluding any SPARQL query generation and result processing and network related processing.

Hence, we can consider the first two categories to be the IO overhead involved in performing Natural Language queries using our semantics.

A 15 megabit cable connection was used during these tests. Before performing any experiments, we measured latency to the remote SPARQL triplestore as being approximately 60 milliseconds.

6.4.1 Results

Profiling for IO:

- For query 1: approximately 98.3% of the running time was spent waiting on IO
- For query 2: approximately 98.5% of the running time was spent waiting on IO

Profiling for CPU time (excluding IO time):
Running time breakdown for query 1:

- SPARQL query generation and result processing including XML parsing: 51.4%
- Network related processing: 39.3%
- Natural Language parsing: 5.5%
- Semantic functions: 3.8%

Running time breakdown for query 2:

- SPARQL query generation and result processing including XML parsing: 42%
- Network related processing: 44%
- Natural Language parsing: 3%
- Semantic functions: 11%

6.4.2 Discussion

As is evident from the results, our implementation is heavily IO bound. The vast majority of time is spent performing the SPARQL queries themselves, with less than 2% of CPU time actually spent in our semantic functions. This means that improvements to the time complexity of the semantic functions has little impact in the overall responsiveness of the system.

This provides a good hint as to how to improve the semantics in the future. We could potentially alleviate this bottleneck by performing fewer external triplestore queries, either by using more advanced forms of query fusion or better caching mechanisms. Such techniques are discussed further in Chapter 9.

Chapter 7

Proof of the Thesis

In Chapter 3, we described a new event-based denotational semantics that improves on the work described by Frost et al. in 2013[16] and Frost and Agboola in 2014[10]. In Chapter 4, we described our modifications to the XSaiga parser that enables semantic functions to be non-referentially transparent, suitable for querying external triplestores. This parser allows both the syntax and the semantics of the Interface to be defined together in the attribute grammar[25], improving modularity, and new semantic rules can be easily and naturally coded in an attribute grammar that supports left recursion, improving extensibility. In Chapter 5, we showed how this new semantics can be implemented efficiently in Haskell and integrated with a parser constructed as an attribute grammar in order to handle Natural Language queries. In Chapter 2, we demonstrated our semantics in action by creating two Web-based interfaces to interact with our query program, operating directly on the Semantic Web.

Therefore, we have shown that by integrating a novel event-based denotational semantics with a parser constructed as an executable attribute grammar, it is possible to create a highly modular and extensible Natural Language Interface to the Semantic Web that supports the use of prepositional phrases in queries.

Chapter 8

Conclusions

In conclusion, we have shown that it is possible to create a highly modular and extensible Natural Language Interface to the Semantic Web that supports the use of prepositional phrases in queries using our approach.

We presented a novel event-based denotational semantics, UEV-FLMS, that improves on EV-FLMS, unifying the treatment of several semantic concepts, solving two problems with the original semantics, and expanding the capabilities of prepositional phrases. In addition to this, we demonstrated a novel way of handling the word "by", as in "discovered by" in our semantics by treating it directly as a preposition. We integrated this semantics with a parser constructed as an executable attribute grammar, extending the work by Frost and Hafiz in 2008[25] to support monadic values. We showed that our approach was viable by performing two benchmarks in Chapter 6, and improving on the asymptotic time-complexity of the original semantics with our `condense` function. We discussed potential methods to improve efficiency further in Chapter 9. We also uploaded our work to Hackage, an online repository of Haskell packages, in the form of the XSaiga package. Finally, we built an online query interface to this program, creating a highly modular and extensible Natural Language Interface to the Semantic Web that supports complex chained prepositional phrases in queries.

The approach used in this thesis for handling the word "by" could potentially apply to other related problems in Natural Language Processing. In unifying the treatment of distinct semantic concepts, it may be possible to find simpler ways of handling linguistic

concepts which have been inherently difficult to capture in formal semantics. It also could be used to cleanly handle the separation of Primary and Secondary sources, and future semantics may be able to infer the truth of statements by agreement among Secondary sources. For example, in the statement “Sally said that John thinks the moon is made of cheese”, In the absence of the definitive statement to the contrary from a Primary source, e.g. “John denied thinking that the moon is made of cheese”, it may be reasonable to infer that John believes that the moon is made of cheese, especially if Sally is a particularly trustworthy source. Research in this area will become increasingly important in order to assess the trustworthiness of results from queries in the Semantic Web. It may also be possible to apply our technique for handling prepositional phrases to handling language constructs seen in other languages, such as postpositions and circumpositions.

One area where our research could be particularly relevant is in constructing Natural Language Interfaces for IoT-enabled devices. It could be feasible to provide an interface to control a variety of these devices using our apporach, improving accessibility for users who suffer disabilities. As the Semantic Web becomes more mainstream, there will be an increasing need for enabling technologies like these. It is our hope that researchers in the future will consider building on our approach in order to fulfill this growing need.

Chapter 9

Future Work

9.1 Providing Event-based views into entity-based triplestores

The system presented in this thesis report requires event-based triplestores in order to function. Much of the Semantic Web, however, is not comprised of event-based triplestores. In order to perform queries on these databases, there must be a way to transform these existing triplestores into an event-based form or provide an event-based "view" into these databases.

In practice, entity-based triplestores also contain *ontology* information that describes the structure of particular sets of triples. An ontology in the Semantic Web serves a similar purpose as a schema does in a relational database. In fact, the original language proposed by the W3C for describing ontologies was called *RDF Schema*. This evolved into what is known today as the *Web Ontology Language* or *OWL* for short.

Definition 19 (Ontology)**.** *"An ontology is an explicit specification of a conceptualization"[36]*

Using information present in Web Ontologies, it may be possible to provide event-based views into entity-based data.

9.2 Thoughts on scaling up to handle massive triplestores

There are several drawbacks with the current implementation that prevent it from being used with massive triplestores. In the worst case, some of the functions would require reading in a significant amount of data from the triplestore in order to return a value. One example of this is in the membership functions.

In addition to this, the semantics as they exist currently in some cases perform many small queries to the triplestore, slowing down processing dramatically. In particular, as seen in Chapter 6, running time of the semantics was empirically measured to be dominated by the IO involved in actually communicating with the remote triplestore.

An example of where these small queries are made are in the `make_trans_active'` and `make_trans_passive'` semantic functions.

In these functions, `getts_entevprop` is applied to each event list inside the FDBR passed to the function for each preposition's properties. `getts_entevprop` makes a request to the remote triplestore for each event list. Therefore, if there are n FDBR-pairs inside the FDBR, and m prepositions, $n * m$ requests will be made to the remote triplestore in the worst case.

9.2.1 Query fusion

One way of remedying this is to reduce the number of queries to the remote triplestore. To achieve this, it may be possible to modify the semantics to support *query fusion*, fusing smaller queries together into larger queries in order to reduce the number of queries performed.

In Chapter 4 of this Thesis Report, it was explained that the parser now operates in the IO monad, and that with a small amount of work, it could work in other monads as well. To support query fusion in the semantics, a new monad could be devised that the semantic functions would use instead of the IO monad. Let us call this hypothetical monad `QueryFusion`. This monad would be pure, functioning much like the monad `State` as it exists in the Haskell language currently: threading an implicit *state* argument through the combinators and semantic functions and keeping the details of that state neatly abstracted

away. The goal of this would be to obtain a value of type `QueryFusion a`, which with the help of a function, say, `runQueryFusion`, that would convert the query into an IO action.

One nice side effect of this is that it would allow the semantics to once again be defined as pure functions.

An example of how this could work:

```
runQueryFusion :: QueryFusion a → IO a


main = do
  rawQuery ← getQueryString      -- rawQuery :: String
  let sQuery = genQuery rawQuery  -- sQuery   :: QueryFusion Result
  result ← runQueryFusion query  -- result   :: Result
  print result
```

The actual optimization itself would occur in the `runQueryFusion` function. A variety of transformations could occur in this process.

A basic form of query fusion exists in the Solarman source code currently that relies on *memoization*. The SPARQL backend in the Getts module remembers the results of previous queries, and so if another query is made for the same information, the previous result is returned. This optimization was made under the assumption that the triples in the triplestore would not change in the span of time that the query was being made. More sophisticated query fusions would be implementable with the above scheme. It may be feasible, for instance, to preprocess queries with this method to form one large "super-query" to the endpoint that contains all triples needed by each semantic function used in the query. This would eliminate the overhead associated with performing many small queries to remote SPARQL endpoints, as semantic functions could operate directly on local memory.

9.2.2 Data parallelism

Assuming that the Query Fusion changes are implemented, further optimizations could occur by exploiting parallelism within the semantic functions. Currently, no parallelism is taken advantage of within the semantics, as they are implemented in a single-threaded manner.

Definition 20 (Task parallelism)**.** *Dividing a problem into independent* tasks *and executing them in parallel, possibly on different sets of data*

Semantic functions which do not depend on one another, for example `moon` and `planet`, could be evaluated in parallel to accelerate processing. In addition to this, when the final FDBRs are obtained for the semantic functions, *data parallelism* could be used to efficiently perform operations on FDBRs.

Definition 21 (Data parallelism)**.** *A special case of task parallelism where the tasks are identical, but are executed over different sets of data*

As an example of data parallelism, consider the problem of squaring each integer in a list. Let us define a function `square` that computes the square of a number.

```
square :: Integer → Integer
square x = x * x
```

A single-threaded Haskell program might evaluate `square` across all of the elements in a list in the following manner:

```
list :: [Integer]


squareList = map square


> squareList [1,2,3] ⇒ [1,4,9]
```

Note that squaring one number in the list does not depend on the square of any other number in the list. Therefore, we could in theory compute the elements of `squareList [1,2,3]` in parallel.

Suppose we have n hardware threads of execution available to use. Let us define a

function to partition `list` into *n* sublists:

```
partition n = foldr buildPart (empty n) ∘ split n


split n [] = []
split n list = (take n list) : (split n $ drop n list )


empty 0 = []
empty n = [] : (empty (n - 1))


buildPart x partition
  = (zipWith (:) x partition) ++ drop (length x) partition
```

Now, execute `square` over each sublist with a hypothetical `parMap` function which performs a `map` across each element in a list in parallel, distributing each task among a different core:

```
result = concat $ parMap squareList $ partition n list
```

In this view, `squareList` is a task being executed in parallel across all of the partitions of our input list. Specifically, this example is exhibiting data parallelism, as we have one task `squareList` that is being executed over different sets of data. The `concat` function merges the results back into one list.

One benefit of data parallelism is that it maps well onto a variety of different compute architectures, such as FPGAs, GPUs, and compute clusters. In massive triplestores, it may not be feasible to perform computations on FDBRs in a reasonable amount of time using only single-threaded semantics, as in these triplestores, an FDBR could contain millions of FDBR-pairs. Fortunately, each FDBR-pair exists independent of other FDBR-pairs, and the semantic functions could be rewritten in a data-parallel way similar to how the above example was rewritten in order to scale across the available hardware threads in a system. In doing so, the door would be open for implementing the semantics on GPUs as well.

9.2.3 Conceptual spaces

One promising approach to processing large amounts of data involves the use of Conceptual Spaces[31][32][13][9], which has already seen some use in performing queries in the Semantic Web[28][24][13]. It may be possible to develop a new event-based semantics that uses Conceptual Spaces, and by extension Conceptual Geometry, to perform queries on larger datasets.

9.3 Summary

In this chapter we discussed two potential avenues for future work. We first discussed the notion of an event-based view into an entity-based triplestore. We then discussed potential ways to improve the performance of our semantics, through more advanced forms of query fusion and taking advantage of the data-parallel aspects of FDBR filtering in our semantics. In the remaining chapters of this Thesis, we present the proof of our Thesis Statement and discuss conclusions that could be drawn from our work.

Bibliography

[1] AChecker. *IDI Web Accessibility Checker*. http://achecker.ca/checker/index.php. [Online; accessed 11-November-2016]. 2016 (cit. on p. 13).

[2] S. Peelar. *Solarman Natural Language Interface*. http://speechweb2.cs.uwindsor.ca/solarman2/demo_sparql.html. [Online; accessed 11-November-2016]. 2016 (cit. on p. 6).

[3] S. Peelar. *XSaiga Haskell Package*. https://hackage.haskell.org/package/XSaiga. [Online; accessed 11-November-2016]. 2016 (cit. on pp. iv, 13).

[4] *Prelude*. https://hackage.haskell.org/package/base-4.9.0.0/docs/Prelude.html. [Online; accessed 28-November-2016]. 2016.

[5] *Top level mutable state*. https://wiki.haskell.org/Top_level_mutable_state. [Online; accessed 28-November-2016]. 2016 (cit. on pp. 47, 49).

[6] WebAIM. *WAVE Web Accessibility Evaluation tool*. http://wave.webaim.org/. [Online; accessed 11-November-2016]. 2016 (cit. on p. 13).

[7] W. Agboola. "An extensible natural-language query interface to the DBpedia Triple-store". M.Sc. Thesis. University of Windsor, 2015 (cit. on pp. 22, 23, 47, 50, 57).

[8] R. E. Kent. "The ERA of FOLE: Foundation". In: *arXiv preprint arXiv:1512.07430* (2015) (cit. on p. 2).

[9] F. Zenker and P. Grdenfors. "Applications of Conceptual Spaces". In: (2015) (cit. on p. 70).

[10] R. A. Frost, W. Agboola, E. Matthews, and J. A. Donais. “An Event-Driven Approach for Querying Graph-Structured Data Using Natural Language”. In: *EDBT/ICDT Workshops*. Vol. 2014. 2014, pp. 192–199 (cit. on pp. iii, 5, 18, 24, 62).

[11] R. A. Frost, J. Donais, E. Mathews, W. Agboola, and R. Stewart. “A Demonstration of a Natural Language Query Interface to an Event-Based Semantic Web Triplestore”. In: *ESWC (Satellite Events)*. Springer LNCS Volume 8798. 2014, pp. 343–348 (cit. on pp. iii, iv, 5, 24, 44).

[12] R. A. Frost, J. Donais, E. Matthews, and R. Stewart. “A denotational semantics for natural langauge query interfaces to semantic web triplestores”. In: *Submitted for publication* (2014) (cit. on pp. 18, 20–22, 25).

[13] P. Grdenfors. *The geometry of meaning: Semantics based on conceptual spaces*. MIT Press, 2014 (cit. on p. 70).

[14] T. W. W. W. C. (W3C). *RDF 1.1 N-Triples*. `https://www.w3.org/TR/n-triples/`. [Online; accessed 11-November-2016]. 2014 (cit. on pp. 2, 14).

[15] T. W. W. W. C. (W3C). *RDF 1.1 Semantics*. `https://www.w3.org/TR/rdf11-mt/`. [Online; accessed 06-September-2016]. 2014 (cit. on p. 2).

[16] R. A. Frost, B. S. Amour, and R. Fortier. “An Event Based Denotational Semantics for Natural Language Queries to Data Represented in Triple Stores”. In: *ICSC, 2013 IEEE Seventh International Conference on Semantic Computing*. IEEE. 2013, pp. 142–145 (cit. on pp. 5, 18, 27, 62).

[17] J. M. Jansen. “Programming in the λ-calculus: From Church to Scott and back”. In: *The Beauty of Functional Code*. Springer, 2013, pp. 168–180 (cit. on p. 51).

[18] A. Jena. “Apache jena”. In: *jena. apache. org [Online]. Available: http://jena. apache. org [Accessed: Mar. 20, 2014]* (2013) (cit. on p. 14).

[19] D. R. Dowty, R. Wall, and S. Peters. *Introduction to Montague semantics*. Vol. 11. Springer Science & Business Media, 2012 (cit. on p. 5).

[20] D. Terei, S. Marlow, S. Peyton Jones, and D. Mazires. “Safe Haskell”. In: *ACM SIGPLAN Notices*. Vol. 47. 12. ACM. 2012, pp. 137–148 (cit. on p. 10).

[21] J. Lehmann and L. Bhmann. “Autosparql: Let users query your knowledge base”. In: *Extended Semantic Web Conference*. Springer. 2011, pp. 63–79 (cit. on p. 4).

[22] M. Lipovaca. *Learn You a Haskell for Great Good!: A Beginner’s Guide*. no starch press, 2011 (cit. on pp. 23, 39).

[23] O. Erling and I. Mikhailov. “Virtuoso: RDF support in a native RDBMS”. In: *Semantic Web Information Management*. Springer, 2010, pp. 501–519 (cit. on pp. 9, 14).

[24] B. Adams and M. Raubal. “Conceptual Space Markup Language (CSML): Towards the Cognitive Semantic Web.” In: *ICSC*. 2009, pp. 253–260 (cit. on p. 70).

[25] R. A. Frost, R. Hafiz, and P. Callaghan. “Parser combinators for ambiguous left-recursive grammars”. In: *International Symposium on Practical Aspects of Declarative Languages*. Springer LNCS Volume 4902. 2008, pp. 167–181 (cit. on pp. iii, iv, 5, 8, 37, 38, 41, 54, 62, 63).

[26] V. Tablan, D. Damljanovic, and K. Bontcheva. “A natural language query interface to structured information”. In: *European Semantic Web Conference*. Springer. 2008, pp. 361–375 (cit. on p. 4).

[27] P. Cimiano, P. Haase, J. Heizmann, and M. Mantel. *Orakel: A portable natural language interface to knowledge bases*. Tech. rep. Technical report, Institute AIFB, University of Karlsruhe, 2007 (cit. on p. 4).

[28] X. Wu, L. Zhang, and Y. Yu. “Exploring social annotations for the semantic web”. In: *Proceedings of the 15th international conference on World Wide Web*. ACM. 2006, pp. 417–426 (cit. on p. 70).

[29] T. W. W. W. C. (W3C). *Uniform Resource Identifier (URI): Generic Syntax*. `https://tools.ietf.org/html/rfc3986`. [Online; accessed 11-November-2016]. 2005 (cit. on p. 2).

[30] G. Antoniou and F. Van Harmelen. *A semantic web primer*. MIT press, 2004 (cit. on p. 17).

[31] P. Grdenfors. *Conceptual spaces: The geometry of thought*. MIT press, 2004 (cit. on p. 70).

[32] D. Widdows and D. Widdows. *Geometry and meaning*. Vol. 773. CSLI publications Stanford, 2004 (cit. on p. 70).

[33] J. Broekstra and A. Kampman. "SeRQL: a second generation RDF query language". In: *Proc. SWAD-Europe Workshop on Semantic Web Storage and Retrieval*. 2003, pp. 13–14 (cit. on p. 4).

[34] S. Palmer. *The semantic web: An introduction*. `http://infomesh.net/2001/swintro/`. [Online; accessed 28-November-2016]. 2001 (cit. on p. 1).

[35] G. Hutton and E. Meijer. "Monadic parsing in Haskell". In: *Journal of functional programming* 8.04 (1998), pp. 437–444 (cit. on p. 37).

[36] T. R. Gruber. "Toward principles for the design of ontologies used for knowledge sharing?" In: *International journal of human-computer studies* 43.5 (1995), pp. 907–928 (cit. on p. 65).

[37] G. T. Leavens. "A Physical Example for Teaching Curried Functions". In: (1995) (cit. on p. 23).

[38] R. A. Frost. "Constructing programs as executable attribute grammars". In: *The Computer Journal* 35.4 (1992), pp. 376–389 (cit. on p. 5).

[39] R. Frost and J. Launchbury. "Constructing natural language interpreters in a lazy functional language". In: *The Computer Journal* 32.2 (1989), pp. 108–121 (cit. on pp. 5, 8, 41).

Appendices

Appendix A - Source code

The source code for Solarman and the XSaiga parser can be obtained online via this URL:

`https://hackage.haskell.org/package/XSaiga-1.5.0.0/XSaiga-1.5.0.0.tar.gz`

The XSaiga package for Haskell is available online at this URL:

`https://hackage.haskell.org/package/XSaiga`

Vita Auctoris

Shane Peelar was born in 1990 in Windsor, Ontario. He completed his undergraduate degree in Computer Science from the University of Windsor in 2014, graduating with Honours and specializing in Software Engineering. He then went on to complete his Masters degree in Computer Science from the University of Windsor in 2016.