CHALMERS
UNIVERSITY OF TECHNOLOGY

Semantic verification of multilingual texts

A project of textual verification using Grammatical Framework

Daniel Stribrand

12 September 2022

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Master's thesis 2022

Semantic verification of multilingual texts

A project of textual verification using Grammatical Framework

Daniel Stribrand

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2022

Semantic verification of multilingual texts
A project of textual verification using Grammatical Framework
Daniel Stribrand

Supervisor: Aarne Ranta, Computer Science and Engineering
Examiner: Krasimir Angelov, Computer Science and Engineering

Master's Thesis 2022
Department of Computer Science and Engineering

Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Gothenburg, Sweden 2022

Semantic verification of multilingual texts
A project of textual verification using Grammatical Framework
Daniel Stribrand
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

Verifying text, both syntactically and semantically, is a very necessary task of text generation. By treating languages formally, semantic verification can be done computationally.

With Grammatical Framework, grammars can be defined for multilingual text generation. By using these defined grammars, we can parse text into abstract syntax trees which can be interpreted as first-order logic formulas using the theories of Montague grammar & semantics. These formulas can thereafter be processed and transformed into intermediate representations such that they can be verified against data.

In this project, we have successfully managed to create a program which can fact check simple textual statements about geography against data from WikiData, using a small Grammatical Framework grammar that is interpreted in accordance to Montague grammar & semantics.

Keywords: Grammatical Framework, natural languages, formal languages, text, Montague, grammar, semantics, functional programming, verification, fact checking.

Acknowledgements

I am very grateful for my supervisor Aarne Ranta who helped me navigate through the early process of the project by introducing me to Montague grammar & semantics, Grammatical Framework and how we could use a Grammatical Framework grammar with Haskell.

Daniel Stribrand, Gothenburg, September 2022

Contents

1

Introduction

Say that you are given the statement

```
The population of Sweden is about 10 million.
```

and you are tasked to verify it. A very natural approach to this would be to 'look-it-up' in an encyclopedia. For a single sentence like this, such an approach is very accessible. But what if you are tasked to verify a much larger amount of sentences? Another approach is to verify the statement by comparing it to data computationally. Consider the idea that you have access to a data file with the following information about the country Sweden:

```
name: Sweden,
capital: Stockholm,
population: 10402070,
...
```

We can take the studied sentence and compare the information it asserts to the presented data. This approach is pleasant, because it can be automated. However it does pose some problems:

- How would you determine that the population property of Sweden is the one which is to be compared?
- How would you determine that the comparison should be $10000000 \approx 10402070$?

These problems are at the heart of this project, which has been conducted in order to investigate the usage of a system for automatically verifying encyclopedic text. This can be achieved by using Grammatical Framework (a special-purpose functional programming language for grammar generation), Montague grammars & semantics (a formal semantics which represent grammatical categories as first-order logic formulas), and WikiData (the database behind Wikipedia).

1.1 Background

Wikipedia is a user-generated encyclopedia, with articles written by users and facts defined by users — attempting to encapsulate the sum of all knowledge and make

it readily available in all languages. Sadly however, some subjects are given little to no attention and most languages are underrepresented. Therefore, an attempt to mend this is by generating articles automatically through the usage of data and multilingual text robots. For this purpose Grammatical Framework can be used to define grammars of articles, which can be implemented multilingually, and generated with data from WikiData.

In this project however, rather than working with generation of articles, we investigate the ability to automatically verify the semantics of the generated articles by comparing it to data. A project like this is a very central concept for article generation, and if achieved successfully, could sow many benefits.

1.2 Motivations

By computationally verifying generated texts based on predefined grammars, we acquire the following merits:

- The verification can be used to identify incorrect grammars, *i.e.* if the grammar generates incorrect formulas.
- The ability to automatically identify texts with outdated information.
- If a text was generated with a different data source than the one used for verification, then the verification can be used to identify mismatches in the stated information.
- If an independent author writes a text that aligns with the grammar (of which there could be a large library), then that text can be validated automatically.
- The ability to identify ambiguities in texts.

1.3 Scope

At the end of this project we will have developed a program that can take an English or a Swedish text corresponding to a Grammatical Framework grammar, parse that into the corresponding abstract syntax tree, interpret those as first-order logic formulas, and translate the formulas into descriptions which can be compared to data queried from WikiData.

1.3.1 Delimitation

Notably, the grammar will be a considerably small grammar about geographical facts, however it should conceptually cover a large range of ideas which can easily be expanded upon.

1.3.2 Goals

The first goal is to define a Grammatical Framework grammar for geographical facts in English and Swedish.

Then the following goals of this project correspond nicely to the steps of the verification:

- Parse text into abstract syntax trees using Haskell with the defined Grammatical Framework grammar.
- Interpret the abstract syntax trees as first-order logic formulas using Montague grammar & semantics.
- Translate the formulas into description(s).
- Compare the description(s) to descriptions of data queried from WikiData.
- Make an assessment based on the comparison.

2

Theory

In this project we will study the matter of representing natural languages formally, such that the grammar can be verified. For this purpose we will explain what it means for a language to be formal, define the needed logic, and show how languages can be interpreted logically.

Although this project is not a study of programming languages, a section has still been dedicated for that purpose since it acts like a nice stepping stone.

Finally, if you are unfamiliar with Haskell-style function notation, then it would advisable to read Section 3.1 first.

2.1 Formal languages

A language can be seen as the set of all grammatically correct sentences [1], *i.e.* the set of all sentence which are well-formed in accordance to the language's rules for sentence construction. A formal language is a language whose set of grammatically correct sentences all have precisely defined meanings [2]. This is a very clinical perspective of languages, which allows us to treat natural languages as no different from mathematical languages or programming languages [3], assuming that we can give meaning to all of the grammar.

When a language is formalized it can be used computationally for various tasks (*e.g.* verification or inference) [1].

2.2 Mathematical functions

A function $a \to b$ is a computation from its domain of type a to its range of type b. Conceptually, *all* elements in type a will have some mapping to *some* element in type b [1].

A function of type $a \to b$ can be referred to as a first-order quantification, *i.e.* a quantification over the elements in its domain. Furthermore, higher-order functions *e.g.* $(a \to b) \to c$, range over a domain of functions, thus creating a higher-order

quantification [4].

To define functions one can use the λ-calculus notation [1][5]. Without going into detail, consider the function

$$f(x) = 2x + 1$$

which using the λ-notation can be written as

$$\lambda x \mapsto 2x + 1$$

These functions are commonly referred to as anonymous functions, and are very handy since they are really expressive and flexible (notably, they permit functions to be defined inside other functions).

Since both Haskell and Montague grammar & semantics are based on typed λ-calculus [1], they will frequently appear throughout the report.

2.3 Mathematical logic

Broadly speaking, mathematical logics are formal languages that can be used for reasoning about declarative problems [6]. The focus in this project will primarily be on the first-order predicate logic, which is a symbolic logic that can be used for modelling most declarative sentences [6].

2.3.1 Propositional logic

A proposition is a statement that either holds or does not hold, *i.e.* is true or false. The most basic propositions are atoms, *i.e.* statements that are indivisible, *e.g. 'It is raining'*. By combining atoms, new more expressive propositions can be created, which in turn can also be combined [6]. This creates a recursive structure which can be used for composing arbitrary well-formed propositions.

The logic can be defined recursively as follows [6]:

1. All atoms p, q, r and so on are well-formed propositions that are either true or false.
2. Any two propositions P, Q, R and so on can be bound using the connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ and $\leftrightarrow$ to create formulas which are also well-formed propositions.

The following table shows the connectives and their usage:

	Syntax		Semantics
Negation:	$\neg P$	$\Leftrightarrow$	*'not P'*
Conjunction:	$P \wedge Q$	$\Leftrightarrow$	*'P and Q'*
Disjunction:	$P \vee Q$	$\Leftrightarrow$	*'P or Q'*
Implication:	$P \rightarrow Q$	$\Leftrightarrow$	*'if P then Q'*
Equivalence:	$P \leftrightarrow Q$	$\Leftrightarrow$	*'Q if and only if P'*

Table 2.1: The connectives in propositional logic.

Propositional logic is very useful for an abundance of tasks that need to reason about declarative sentences, however it lacks the ability to formulate propositions that reason about values and sets of values [6].

2.3.2 First-order predicate logic

First-order predicate logic extends the propositional logic by introducing individuals, predicates, functions, identities, relations and first-order quantifications. With these, propositions can now reason about properties of individuals, values of individuals and universes of individuals [6].

The predicate logic can be defined recursively in extension to the propositional logic as follows [6]:

1. All individuals (*e.g.* 3, 'Sweden', ...) are well-formed terms.
2. All variables x, y, z and so on are well-formed terms.
3. Any n-place function $f(t_1, ..., t_n)$ where t_1, t_2, ..., t_n are terms is also a well-formed term.
4. Any two terms bound by the identities $=$ and $\neq$ form a well-formed proposition.
5. Any two terms from the same ordering bound by the relations $<$, $>$, $\leq$ and $\geq$ form a well-formed proposition.
6. Any n-place predicate $P(t_1, ..., t_n)$ where t_1, t_2, ..., t_n are terms is a well-formed proposition.
7. All $\forall$-bindings which bind a variable x to some proposition are well-formed propositions.
8. All $\exists$-bindings which bind a variable x to some proposition are well-formed propositions.

The following table shows the terms:

	Syntax		Semantics
Individuals:	3, *'Sweden'*, ...	$\Leftrightarrow$	*A specified member of the universe*
Variables:	x, y, z, ...	$\Leftrightarrow$	*A variable member of the universe*
Functions:	$f(t_1, ..., t_n)$	$\Leftrightarrow$	*A mapping from n members of the universe to another member*

Table 2.2: The terms in first-order logic.

The following table shows the propositions:

	Syntax		Semantics
Identities:	$x = y$, $x \neq y$	$\Leftrightarrow$	*'x is/is not y'*
Relations:	$x < y$, ...	$\Leftrightarrow$	*'x is smaller than y'*
Predicates:	$P(x)$, $Q(y, z)$, ...	$\Leftrightarrow$	*'x has the property P', 'y has the property Q when paired with z'*
Universal quantification:	$\forall x[...x...]$	$\Leftrightarrow$	*'all x are...'*
Existential quantification:	$\exists x[...x...]$	$\Leftrightarrow$	*'some x is...'*

Table 2.3: The propositions in first-order logic.

The main benefit of first-order logic in contrast to propositional logic is the ability to reason about properties of individuals and the ability to reason about universes of individuals.

2.4 Programming languages

Universal computing machines are machines which can compute any computable problem given a specification of the problem [7]. This specification can be given as punch cards, bit sequences or any other appropriate way of conveying mathematical information. However, it is suitably done with a programming language designed for the task [8].

A programming language is in essence a set of rules detailing how to interpret a valid string of symbols, in order to form a program. The validity of the form of the string is decided by the syntax and the interpretation of the string is decided by the semantics of the syntax [8].

2.4.1 Syntax

The syntax is a set of grammar rules which can be used for parsing arbitrary strings into programs (*e.g.* as lists of machine instructions) [8].

The grammar rules are normally defined as context-free function rules of the form

$$C.\ T \leftarrow t_1\ t_2\ ...\ t_n$$

where C is the category of the function and T is the function label. The category is produced by the given sequence of terminals or categories t_1 to t_n. Terminals are elementary symbols (*e.g.* '1', '+', '-') [8]. This type of grammar is also called phrase-structure grammar.

A simple grammar for arithmetical expressions could thus be given as follows with these production rules[1]:

```
Expr.   Par    ←  '(' Expr ')'
Expr.   Div    ←  Expr '/' Expr
Expr.   Mul    ←  Expr '*' Expr
Expr.   Add    ←  Expr '+' Expr
Expr.   Sub    ←  Expr '-' Expr
Expr.   Const  ←  Num

Num.    N2     ←  Digit Num
Num.    N1     ←  Digit

Digit.  Nine   ←  '9'
Digit.  ...    ←  ...
Digit.  One    ←  '1'
Digit.  Zero   ←  '0'
```

Figure 2.1: Small grammar for arithmetical expressions.

By parsing a string using these rules, we can construct an abstract syntax tree that represents the structure of function applications which make out the specification of the expression [8].

As an example, consider the expression

```
1 + (2 * 3)
```

which would be parsed into the following abstract syntax tree of category `Expr`

```
Add (Const (N1 One))
(Par (Mul (Const (N1 Two)) (Const (N1 Three))))
```

of which an interpreter could easily trace the structure of the expression and get the correct interpretation.

2.4.2 Semantics

Similar to the syntax, the semantics are a set of recursive interpretation rules (or functions) corresponding to the categories in the syntax, such that when an abstract syntax tree is passed to the interpretation, it will be unfolded and interpreted node-by-node [8].

For the previous grammar, we can take a given abstract syntax tree and pass it to

[1]For simplicity, assume that higher precedence is given to rules presented further up.

the following recursive interpretation functions[2]:

```
interpret: Expr → Float
interpret (Par e)            =  interpret e
interpret (Div e1 e2)        =  interpret e1 / interpret e2
interpret (Mul e1 e2)        =  interpret e1 * interpret e2
interpret (Add e1 e2)        =  interpret e1 + interpret e2
interpret (Sub e1 e2)        =  interpret e1 - interpret e2
interpret (Const n)          =  interpret n

interpret: Num → Integer
interpret (N2 d n)           =  interpret d ++ interpret n
interpret (N1 d)             =  interpret d

interpret: Digit → Integer
interpret (Nine)             =  9
interpret (...)              =  ...
interpret (One)              =  1
interpret (Zero)             =  0
```

Figure 2.2: Interpretation of the grammar for expressions.

This example shows that it is a simple matter of unfolding each part of the abstract syntax tree, match it with the appropriate left-hand expression, and evaluate it as the corresponding right-hand expression.

Naturally, as the grammar gets more complex, the interpretations will also be more demanding. That said, conceptually it still is just a matter of applying a recursive interpretation function on the tree, in order to unfold the computation.

2.5 Natural languages

Working with natural languages formally is not very different from working with programming languages. Essentially, you will still be working with syntax (of which we will refer to as grammar from here on) and semantics of the grammar (*i.e.* the interpretations). The striking difference is that grammars are much more complex and the semantics are much more ambiguous.

2.5.1 Grammar

Grammars of natural languages are defined using grammatical categories, which can be combined into larger structures in accordance to the grammar rules. These categories include, but are not limited to, the following:

[2]For simplicity, assume that `++` simply appends an integer in front of another integer (*e.g.* `12 ++ 34 = 1234`).

Category		**Example**
Sentences	`S`	*e.g. 'Daniel is writing on a report every day.'*
Noun phrases	`NP`	*e.g. 'a report', 'Daniel'*
Common nouns	`CN`	*e.g. 'report', 'day'*
Verb phrases	`VP`	*e.g. 'is writing'*
Determiners	`DET`	*e.g. 'a', 'every'*
...	...	...

Table 2.4: Some grammatical categories in natural languages.

Just like with the programming languages, these can be combined to create syntactic structures in the form of abstract syntax trees using the grammar rules [1]. Just like with the programming languages, we can define a small grammar for English accordingly:

```
S.    NpVp    ← NP VP '.'
S.    NpCn    ← NP 'is' 'a' CN '.'
NP.   DetCn   ← DET CN
NP.   Nemo    ← 'Nemo'
VP.   Swims   ← 'swims'
VP.   Sleeps  ← 'sleeps'
CN.   Fish    ← 'fish'
DET.  Every   ← 'every'
DET.  Some    ← 'some'
```

Figure 2.3: Small English grammar example.

This grammar allows us to construct sentences like

```
Nemo is a fish.
```

which would be parsed as

```
NpCn Nemo Fish
```

or a sentence like

```
Every fish swims.
```

which would be parsed as

```
NpVp (DetCn Every Fish) Swims
```

2.5.2 Semantics

Semantics of natural languages can be defined as interpretations which translate from the grammar of the language into corresponding representations of another formal language [2].

For this we can use Montague grammar & semantics, which is a theory for how natural language grammars can be interpreted as first-order logic formulas [1][4]. The types of these categories are given as follows (we use the type symbols e for terms/individuals and t for propositions):

Category		**Type**
Sentences	`S`	t
Noun phrases	`NP`	$(e \to t) \to t$
Common nouns	`CN`	$e \to t$
Verb phrases	`VP`	$e \to t$
Determiners	`DET`	$(e \to t) \to ((e \to t) \to t)$
...	...	...

Table 2.5: Types of some grammatical categories.

We can use this type information to define overloaded interpretation functions on the previously defined English grammar (here we use the type symbols `E` for terms/individuals and `T` for propositions):

```
interpret: S → T
interpret (NpVp np vp)  =  (interpret np) (interpret vp)
interpret (NpCn np cn)  =  (interpret np) (interpret cn)
```

Figure 2.4: Function for interpreting sentences.

Interpreting sentences is mostly a matter of interpreting its components and applying them.

```
interpret: NP → ((E → T) → T)
interpret (DetCn det cn)  =  (interpret det) (interpret cn)
interpret (Nemo)          =  λP ↦ P('Nemo')
```

Figure 2.5: Function for interpreting noun phrases.

Noun phrases are interpreted as functions which takes a predicate and applies it onto its subject.

```
interpret: VP → (E → T)
interpret (Swims)   =  λx ↦ Swims(x)
interpret (Sleeps)  =  λx ↦ Sleeps(x)
```

Figure 2.6: Function for interpreting verb phrases.

Verb phrases are conversely interpreted as functions which takes an individual and applies its predicate.

```
interpret: CN → (E → T)
interpret (Fish)  =  λx ↦ Fish(x)
```

Figure 2.7: Function for interpreting common nouns.

Common nouns are similar to verb phrases in that they are nouns which describes an individual.

```
interpret: DET → ((E → T) → ((E → T) → T))
interpret (Every)  =  λP ↦ (λQ ↦ ∀x[P(x) → Q(x)])
interpret (Some)   =  λP ↦ (λQ ↦ ∃x[P(x) ∧ Q(x)])
```

Figure 2.8: Function for interpreting determiners.

Finally, determiners are first-order quantifiers which assert that all/some individuals that are P is/does Q.

Using these interpretation rules, every well-formed sentence will be interpreted as a single first-order logic formula. Notably, our two previous examples would give the following formulas:

```
Nemo is a fish     ⇒  Fish('Nemo')
Every fish swims.  ⇒  ∀x[Fish(x) → Swims(x)]
```

Both of which correctly assert the meaning of the sentences logically.

3

Technologies

At the heart of this project we have Grammatical Framework for grammar representation and transformation, however most development behind the verification is done using Haskell. A few words will also be spent on WikiData since it is used as data model for the verification.

3.1 Haskell

Haskell is a general-purpose functional programming language that is very suitable for formal semantics since it is based on the typed λ-calculus [1][9].

One can even go as far as to say that reducing λ-calculus expressions, evaluating Haskell programs, and computational semantics are in essence the same thing [1].

Consider the function

$$f(x) = 2x + 1$$

in Haskell we would write this as

```
f x = 2 * x + 1
```

or alternatively

```
f = \x → 2 * x + 1
```

where the backslash is a place-in for λ. To call the function, we simply apply an argument after the function name, *e.g.*

```
f 5
```

Typing in Haskell is done through the arrow-notation, *e.g.*

```
f: Integer → Integer
```

which tells us that we need an `Integer` to produce another `Integer`.

Furthermore, custom data types can be defined. This is done using constructors which in turn are functions producing values of the declared type. Consider

$$\texttt{data T = C}_1 \texttt{ | C}_2 \texttt{ | ... | C}_n$$

which is a type called `T` with n constructors. Sometimes we want the constructors to require more type information in order to complete the type construction, which we simply can append to the constructor, *e.g.*

$$\texttt{data T = C}_1 \texttt{ Integer String | C}_2 \texttt{ T | ... | C}_n \texttt{ Double}$$

Finally, functions can be declared using pattern matching on constructors, *e.g.*

$$\begin{aligned}
&\texttt{g: T} \rightarrow \texttt{...} \\
&\texttt{g (C}_1 \texttt{ int str)} &&= \texttt{ ... int ... str ...} \\
&\texttt{g (C}_2 \texttt{ t)} &&= \texttt{ ... t ...} \\
&\texttt{g (C}_n\texttt{)} &&= \texttt{ ...}
\end{aligned}$$

where given a value of type `T`, the function `g` will match it against the left-hand patterns in order to evaluate which right-hand expression to use.

3.2 Grammatical Framework

Grammatical Framework is a special-purpose functional programming language for grammars. Conceptually, a project with Grammatical Framework will define an abstract grammar that is to be used for representation of the textual constructs in the grammar's domain, which in turn can be implemented with concrete grammars for each desired language that the text is to be rendered in [10].

3.2.1 Abstract grammars

The abstract grammar is in essence all type declarations of the grammar, which are shared with the concrete grammars, and make up the data structures of the abstract syntax trees.

As an example, consider the following categories: `CN`, `DET` and `NP`. We populate them accordingly:

$$\begin{aligned}
&\texttt{car : CN} \\
&\texttt{the : DET}
\end{aligned}$$

(*i.e.* `car` is a common noun and `the` is a determiner). We can declare a meaning-building function `mkNP` as

$$\texttt{mkNP : DET} \rightarrow \texttt{CN} \rightarrow \texttt{NP}$$

With this a `NP` can be constructed by simply applying `car` and `the` to `mkNP` as

```
mkNP the car
```

3.2.2 Concrete grammars

The concrete grammar dictates how the abstract grammar is to be rendered in a specific language. Here each category is given a linearization corresponding to its grammatical category (*e.g.* noun phrase, common noun, etc.).

The rendering is computed by linearization rules which are function patterns acting as implementations of the abstract grammar. Considering the example presented in the abstract grammar above, we could build a linearization like:

```
mkNP det cn = det ++ cn
```

3.2.3 Resource grammar libraries

To make Grammatical Framework more accessible, a library called the Resource Grammar Library (RGL) has been developed in order to provide a large amount of categories and linearization rules in a range of languages. Therefore, a developer only has to map their chosen categories to the resource grammar, and build their linearizations using the existing linearizations of the categories [10][11].

The Resource Grammar Library is really what gives Grammatical Framework its multilingual capabilities.

3.2.4 Portable Grammar Format

The Portable Grammar Format (PGF) is a representation of Grammatical Framework grammars such that they can be interpreted outside of Grammatical Framework. Notably, an interpreter has been implemented in Haskell [12].

3.3 WikiData

WikiData is the community-created knowledge-graph behind Wikipedia. In essence, it is a large database which can be queried using SPARQL [13]. Notably for this project, it allows the user to query geographical facts.

Similarly to most databases, a query gives a tabular result, where each row is a result of the query and each column is one property.

4

Overview

Before diving into the details, an overview of the project will be provided.

4.1 Development of the grammar

The basis for this project is a Grammatical Framework grammar with English and Swedish linearizations. With this grammar we can express geographical facts about countries and cities, such as *e.g.* statements about capitals, populations, and currencies. In this project, the abstract grammar is of utmost importance, whereas the concrete grammar can be left mostly unattended.

Although the grammar has been modified quite a bit, the starting point was an existing grammar for geographical facts developed by Aarne Ranta[1].

4.2 Interpretation of the grammar

With the grammar complete, the next part is to interpret it as first-order logic using the ways of Montague grammar & semantics. Since Grammatical Framework is not general-purpose, Haskell will have to be used here.

For the interpretation we will have to do three things:

1. Develop a suitable Haskell data type to represent first-order logic formulas.
2. Develop Haskell data types corresponding to the Grammatical Framework abstract grammar functions.
3. Develop interpretation functions of the grammar in Haskell, which interprets the grammar as first-order logic formulas.

[1] See: aarneranta/NLG-examples at GitHub.

4.2.1 Representing Grammatical Framework functions as Haskell data types

To define the interpretation functions in Haskell, it would be suitable if the abstract grammar functions could be represented as Haskell data types, so that the parsing builds abstract syntax trees out of those types and that the interpretation functions can be defined using pattern matching on those types.

As an example, consider a category `N` with the following functions in the abstract grammar

```
X:  A → B → N
Y:      C → N
```

It would be suitable to have a Haskell data type of the form

```
data N = X A B | Y C
```

Since then an interpretation function `interpret` could be defined as

```
interpret: N → ...
interpret (X a b)    =  ... a ... b ...
interpret (Y c)      =  ... c ...
```

in Haskell.

Fortunately, these Haskell data types can be automatically generated from the Grammatical Framework grammar (see Appendix A), and parsing from text into these data types is already available with the Haskell interpreter of PGF.

4.3 Verification of the interpretation

With the grammar interpreted as first-order logic formulas, the final step is to verify it against data. For this we will develop an intermediate representation coined 'descriptions', where the verification will be a comparison between the descriptions generated from the interpretations of the text and descriptions generated from the data model.

The data model will be a list of descriptions which has been generated out of the data queried from WikiData.

4.4 Evaluation

The evaluation of this project will be in the form of a sanity check (every individual in the data model should verify against itself) and verification of known universal geographical facts (*e.g.* the People's Republic of China has the largest population).

5

Project

With the project properly defined, it is now time to present a more detailed account.

5.1 Implementation of first-order predicate logic

The implementation of the first-order logic is intentionally chosen to be a subset of the full logic. This is because the used abstract grammar does not need the full logic to be interpreted, and the usage of a smaller logic simplifies the specification.

5.1.1 Types

In the specification of the logic we will use the following type signatures:

Name	**Symbol**
Individuals/terms:	e
Values:	v
Propositions:	t

Table 5.1: The type signatures used in the specification.

5.1.2 Propositions

A proposition can be defined using the following constructors:

Name	**Form**	**Type**
One-place predicates:	$P(x)$	$e \to t$
Conjunction:	$Q \wedge R$	$t \to t \to t$
Implication:	$Q \to R$	$t \to t \to t$
Universal quantification:	$\forall x[...x...]$	$(e \to t) \to t$
Relations:	$x \diamond y$	$v \to v \to t$

Table 5.2: The propositions used in this project.

(Note that $\diamond$ is used as a placeholder for a binary relation.)

5.1.3 Binary relation operators

Name	Symbol
Equality:	$=$
Greater-than:	$>$
Less-than:	$<$
Greater-than-or-equal:	$\geq$
Less-than-or-equal:	$\leq$
Approximately:	$\approx$

Table 5.3: The binary relations used in the propositions.

Only a few of the common binary relations are needed. Notably, the grammar needs no way to express any negations.

5.1.4 Values

Constructor	Form	Type
One-place functions:	$f(x)$	$e \rightarrow v$
Terms:	*'Sweden'*, ...	v

Table 5.4: The values used in the propositions.

Similarly, the grammar only needs to express relations between values in the form of terms and one-place properties.

5.2 Grammar for geographical facts

The Grammatical Framework grammar is a simple grammar for expressing statements about geography. To do this, there needs to be a way of representing the structure of a text document, how to form sentences, how to express facts, and how to express properties.

The abstract grammar is given a concrete grammar with linearizations in both English and Swedish. However, since the verification only uses the abstract grammar (aside from parsing a string into an abstract syntax tree), the focus will be spent on that.

The categories used in this grammar are the following:

Category	**Linearized as**
`Doc`	Text
`Sentence`	Sentence
`Fact`	Clause
`Object`	Noun phrase
`Property`	Adjectival phrase
`Attribute`	Common noun
`Kind`	Common noun
`Value`	Noun phrase
`Name`	Noun phrase
`Numeric`	Cardinal
`CName`	Noun phrase
`CDName`	Noun phrase or Adjectival phrase

Table 5.5: The categories in the grammar.

How these are used is dictated by the meaning-building functions declared in the abstract grammar.

The categories are combined using the following functions:

Function	Type
OneSentenceDoc:	Sentence → Doc
AddSentenceDoc:	Doc → Sentence → Doc
FactSentence:	Fact → Sentence
ConjSentence:	Sentence → Sentence → Sentence
KindFact:	Object → Kind → Fact
PropertyFact:	Object → Property → Fact
AttributeFact:	Attribute → Object → Value → Fact
MaxObjectAttributeFact:	Object → Attribute → Fact
MinObjectAttributeFact:	Object → Attribute → Fact
MaxObjectKindAttributeFact:	Object → Kind → Attribute → Fact
MinObjectKindAttributeFact:	Object → Kind → Attribute → Fact
NameObject:	Name → Object
PronounObject:	Name → Object
ConjObject:	[Object] → Object
PropertyKind:	Property → Kind → Kind
CountryKind:	Kind
CityKind:	Kind
CurrencyKind:	Kind
InhabitantKind:	Kind
DemonymProperty:	CDName → Property
CapitalAttribute:	Attribute
CountryAttribute:	Attribute
PopulationAttribute:	Attribute
ContinentAttribute:	Attribute
CurrencyAttribute:	Attribute
NumericKindValue:	Numeric → Kind → Value
NameValue:	Name → Value
NumericValue:	Numeric → Value
MkName:	CName → Name
MkContinentName:	CDName → Name
IntNumeric:	Int → Numeric
IntMillionNumeric:	Int → Numeric
IntBillionNumeric:	Int → Numeric
AboutNumeric:	Numeric → Numeric
OverNumeric:	Numeric → Numeric
UnderNumeric:	Numeric → Numeric

Table 5.6: The functions in the abstract grammar.

A text will thus be a recursive document consisting of sentences that express one or more facts. These facts can either express properties about objects or assert values to attributes of objects.

In English, these functions will be linearized accordingly:

Function with parameter(s)	**English linearization**
OneSentenceDoc s:	s.
AddSentenceDoc d s:	d. s.
FactSentence f:	f
ConjSentence s_1 s_2:	s_1 'and' s_2
KindFact o k:	o 'is/are a' k
PropertyFact o p:	o 'is/are' p
AttributeFact a o v:	'the' a 'of' o 'is' v
MaxObjectAttributeFact o a:	o 'has/have the largest' a
MinObjectAttributeFact o a:	o 'has/have the smallest' a
MaxObjectKindAttributeFact o k a:	o 'is/are the' k 'with the largest' a
MinObjectKindAttributeFact o k a:	o 'is/are the' k 'with the smallest' a
NameObject n:	n
PronounObject n:	'it'
ConjObject [o_1,o_2,...,o_n]:	o_1, o_2, ... 'and' o_n
PropertyKind p k:	p k
CountryKind:	'country'
CityKind:	'city'
CurrencyKind:	'currency'
InhabitantKind:	'inhabitant'
DemonymProperty c:	c
CapitalAttribute:	'capital'
CountryAttribute:	'country'
PopulationAttribute:	'population'
ContinentAttribute:	'continent'
CurrencyAttribute:	'currency'
NumericKindValue n k:	n k
NameValue n:	n
NumericValue n:	n
MkName c	c
MkContinentName c	c
IntNumeric i:	i
IntMillionNumeric i:	i 'million'
IntBillionNumeric i:	i 'billion'
AboutNumeric n:	'about' n
OverNumeric n:	'over' n
UnderNumeric n:	'under' n

Table 5.7: The functions with parameters and English linearizations.

With these linearizations, it should be clear how the functions can be linearized as a text asserting various geographical facts.

Finally, there is a large generated lexicon of common names that the grammar needs. This includes names for countries, capitals, currencies and continents (with demonyms).

Function	**Category**	**Text**
...		
Sweden_CName:	CName	'Sweden'
Stockholm_CName:	CName	'Stockholm'
Swedish_krona_CName:	CName	'Swedish krona'
...		
Europe_CDName:	CDName	'Europe' or 'European'
...		

Table 5.8: Lexicon of common names.

To take this all together, consider as an example the following abstract syntax tree of a document with n period-separated sentences

```
AddSentenceDoc (... (AddSentenceDoc (OneSentenceDoc
    <sentence 1>) <sentence 2>)) <sentence n>
```

which would be linearized as

```
<sentence 1> . <sentence 2> . ... <sentence n> .
```

A sentence inside of this document will have at least one fact, consider *e.g.*

```
FactSentence (KindFact <object> <kind>)
```

which would be linearized as

```
<object> is/are a <kind>
```

The object could be that of Sweden, *i.e.* `NameObject (MkName Sweden_CName)`, and appropriately, the kind for this would thus be that of countries, *i.e.* `CountryKind`, which finally gives us

```
FactSentence (KindFact (NameObject (MkName
        Sweden_CName) CountryKind)
```

which would be linearized as

```
Sweden is a country
```

5.3 Interpretation of the grammar

The goal of the interpretation is to translate each abstract syntax tree into a first-order logic proposition. Since pronouns (*e.g.* `it`) introduce ambiguities, multiple propositions might have to be inferred.

5.3.1 Translation of categories into propositions

The type of the interpretation of each category is given as the following:

Category		**Type of interpretation**
`Doc`	Text	t
`Sentence`	Sentence	t
`Fact`	Clause	t
`Object`	Noun phrase	$(e \to t) \to t$
`Property`	Adjectival phrase	$e \to t$
`Attribute`	Common noun	$e \to v$
`Kind`	Common noun	$e \to t$
`Value`	Noun phrase	$v \to t$
`Name`	Noun phrase	$(e \to t) \to t$
`Numeric`	Cardinal	$v \to t$
`CName`	Noun phrase	$(e \to t) \to t$
`CDName`	Noun phrase	$(e \to t) \to t$
`CDName`	Adjectival phrase	$e \to t$

Table 5.9: Interpretation of each category.

The interpretation differs from the Montague grammar & semantics on the interpretation of the attributes, which are treated as relations to values. This ultimately also gives us the odd interpretations of the categories `Value` and `Numeric`.

For each category to be interpreted, there has to be one interpretation per function presented in the grammar. For the following interpretations, each argument to the functions is already interpreted in accordance to its category.

Interpretation of the Doc category

$$\texttt{OneSentenceDoc sentence} \Rightarrow \texttt{sentence}$$
$$\texttt{AddSentenceDoc doc sentence} \Rightarrow \texttt{doc} \wedge \texttt{sentence}$$

A document is just a proposition. Multiple sentences in a document can simply be conjoined in the interpretation.

Interpretation of the Sentence category

$$\texttt{FactSentence fact} \Rightarrow \texttt{fact}$$
$$\texttt{ConjSentence sentence}_1\ \texttt{sentence}_2 \Rightarrow \texttt{sentence}_1 \wedge \texttt{sentence}_2$$

A sentence is just a proposition. Multiple sentences conjoined can simply be conjoined in the interpretation.

Interpretation of the FACT category

```
KindFact object kind                  ⇒  kind(object)
PropertyFact object property          ⇒  property(object)
AttributeFact attribute object value  ⇒  attribute(object) ⋄ value
```

Notably in the case of `AttributeFact`, the relation operator $\diamond$ is derived from the interpretation of the value.

The following functions has to be interpreted as first-order quantifications:

```
MaxObjectAttributeFact object attribute ⇒
             ∀x [attribute(x) ≤ attribute(object)]
MinObjectAttributeFact object attribute ⇒
             ∀x [attribute(x) ≥ attribute(object)]
```

We also have two quantified statements which are also conditional:

```
MaxObjectKindAttributeFact object kind attribute ⇒
        ∀x [kind(x) ⇒ attribute(x) ≤ attribute(object)]
MinObjectKindAttributeFact object kind attribute ⇒
        ∀x [kind(x) ⇒ attribute(x) ≥ attribute(object)]
```

Interpretation of the OBJECT category

```
NameObject name             ⇒  λP ↦ P(name)
PronounObject name          ⇒  λP ↦ P(name)
ConjObject [obj1,...,objn]  ⇒  λP ↦ P(obj1) ∧ ... ∧ P(objn)
```

Since objects are noun phrases, they refer to some noun which has to be applied to some predicate. Note that for the `PronounObject`, `name` will refer to an unknown string `<?>` which has to be inferred later on.

Interpretation of the KIND category

```
PropertyKind property kind  ⇒  λx ↦ property(x) ∧ kind(x)
CountryKind                 ⇒  λx ↦ country(x)
CityKind                    ⇒  λx ↦ city(x)
CurrencyKind                ⇒  λx ↦ currency(x)
InhabitantKind              ⇒  λx ↦ inhabitant(x)
```

Conversely, since kinds are properties they have to be provided an individual to apply their predicate(s) to.

Interpretation of the PROPERTY category

$$\texttt{DemonymProperty cdname} \Rightarrow \lambda\texttt{x} \mapsto \texttt{cdname(x)}$$

Properties also simply apply a predicate to an individual.

Interpretation of the ATTRIBUTE category

$$\begin{array}{lcl}
\texttt{CapitalAttribute} & \Rightarrow & \lambda\texttt{x} \mapsto \texttt{capital(x)} \\
\texttt{CapitalAttribute} & \Rightarrow & \lambda\texttt{x} \mapsto \texttt{country(x)} \\
\texttt{PopulationAttribute} & \Rightarrow & \lambda\texttt{x} \mapsto \texttt{population(x)} \\
\texttt{ContinentAttribute} & \Rightarrow & \lambda\texttt{x} \mapsto \texttt{continent(x)} \\
\texttt{CurrencyAttribute} & \Rightarrow & \lambda\texttt{x} \mapsto \texttt{currency(x)}
\end{array}$$

Unlike the previous properties, attributes are functions (and not predicates) applied to an individual. These are thus assumed to be used in a relation to another value.

Interpretation of the VALUE category

Since both the category `Value` and the category `Numeric` are of type $v \to t$, the interpretation of the numerics simply passes on a value `v` to the interpretation of these.

$$\begin{array}{lcl}
\texttt{NumericKindValue numeric kind} & \Rightarrow & \lambda\texttt{v} \mapsto \texttt{(numeric v)} \wedge \texttt{kind(numeric)} \\
\texttt{NameValue name} & \Rightarrow & \lambda\texttt{v} \mapsto \texttt{v = name} \\
\texttt{NumericValue numeric} & \Rightarrow & \lambda\texttt{v} \mapsto \texttt{(numeric v)}
\end{array}$$

`NumericKindValue` might seem a little strange, but it allows us to interpret sentences such as

$$\texttt{The population of Sweden is about}$$
$$\texttt{10 million inhabitants.}$$

as

$$\texttt{population(Sweden)} \approx \texttt{10000000} \wedge$$
$$\texttt{inhabitant(10000000)}$$

The interpretation of `NameValue` creates an identity relation where some value `v` (*e.g.* a function) is equal the `name`.

Interpretation of the NUMERIC category

Since the structure of the `Numeric` category is recursive, this can be be utilized. The first step is to interpret the relation:

```
AboutNumeric numeric  ⇒  λv ↦ v ≈ numeric'
OverNumeric numeric   ⇒  λv ↦ v > numeric'
UnderNumeric numeric  ⇒  λv ↦ v < numeric'
numeric               ⇒  λv ↦ v = numeric'
```

The last case `numeric` represent the case when the numeric is not given a relation in the text, but just some number, *e.g.*

```
The population of Sweden is 10 million.
```

Then the `numeric'` has to be interpreted, which is done in accordance to:

```
IntNumeric int         ⇒  int
IntMillionNumeric int  ⇒  int × 1000000
IntBillionNumeric int  ⇒  int × 1000000000
```

Note that the grammar permits crazy sentences such as

```
The population of Sweden is under over about 10 million
                        inhabitants.
```

This behaviour is however undefined in the interpretation.

Interpretation of the NAME category

The interpretation of the category `Name` is the same as the interpretation of both `CName` and `CDName` (except when a demonym is used), thus the interpretation is delegated to their interpretations.

```
MkName cname            ⇒  cname
MkContinentName cdname  ⇒  cdname
```

Interpretation of the CNAME and CDNAME categories

Grammatically theses categories are treated as noun phrases.

```
...
Sweden_CName:          ⇒  λP ↦ P(Sweden)
Stockholm_CName:       ⇒  λP ↦ P(Stockholm)
Swedish_krona_CName:   ⇒  λP ↦ P(Swedish krona)
...
Europe_CDName:         ⇒  λP ↦ P(Europe)
...
```

The exception is when a demonym is needed (*e.g. European*) because that is treated as a common noun:

```
...
Europe_CDName:  ⇒  λx ↦ European(x)
...
```

5.3.2 Naïve context-building

The final step of the interpretation is to build contexts. This is needed whenever a pronoun is used (*i.e.* `it`/`its`), in order to solve the ambiguity of which individual it is referencing. Prior to the interpretation, every single pronoun (which is represented as a meta element in Grammatical Framework) will be replaced with the unknown individual `<?>`[1].

The context-building is done naïvely by simply building a new proposition for every possible contestant of the pronoun, which includes every individual that has been referenced prior to the pronoun in the text. Note that all contexts do not have to be sensible, but only one of them for the interpretation to be sensible.

Consider the example

```
The capital of Sweden is Stockholm. Its population is
                    about 1 million.
```

which would be interpreted as

```
capital(Sweden) = Stockholm ∧
  population(<?>) = 10000000
```

The context-building will give two contexts

```
capital(Sweden) = Stockholm ∧
population(Sweden) = 10000000
```

and

```
 capital(Sweden) = Stockholm ∧
population(Stockholm) = 10000000
```

5.4 Descriptions

Before the verification against the data model, it is useful to transform the propositions into some data structure which is more suitable for comparison against data. For this task, a data structure referred to as descriptions has been developed.

Conceptually descriptions are just a slight remodelling of the propositions.

[1] This is needed because Grammatical Framework can not convert an abstract syntax tree into the Haskell grammatical data types if there are meta elements present.

5.4.1 The Description type

The descriptions consist of two things: Descriptions and properties of descriptions. Specifically, a description can be one of three types of descriptions:

- An object description, *i.e.* an individual with a list of properties.
- A conditional description, *i.e.* a list of descriptions that hold under a condition of a list of other descriptions.
- A universal description, *i.e.* a list of descriptions that hold for all individuals.

Furthermore, an object description has a list of properties, of which they can be one or more of the following three things:

- A concept, *i.e.* some property, *e.g.* `European`.
- An attribute, *i.e.* a property that has a relation to a value, *e.g.* `population = 123456789`.
- A constraint, *i.e.* a property that puts a constraint on an attribute, *e.g.* `population < population(Sweden)`.

Note that there may be multiple attributes of the same name (*e.g.* if a country spans multiple continents).

5.4.2 From proposition to descriptions

Translating from propositions to descriptions is relatively trivial. In essence:

- Every individual `x` becomes an object, *i.e.* `x{...}`.
- Every predicate `P(x)` becomes a concept for the individual, *i.e.* `x{P}`.
- The two propositions in a conjunction are both translated into descriptions and then merged.
- Every implication becomes a conditional description.
- Every universal quantification becomes a universal description.
- Relations are treated from the perspective of the left-hand expression. Functions in relation to values become attributes of the object and functions in relation to other functions become constraints. Relations with non-functions in the left-hand side have undefined translations, however no such propositions are generated from the interpretation.

This is summarized in the following table:

Proposition		Description
$P(x)$	$\Rightarrow$	`x{P}`
$Q \wedge R$	$\Rightarrow$	`[Descriptions]`$_{Q}$ `++ [Descriptions]`$_{R}$
$Q \rightarrow R$	$\Rightarrow$	`[Descriptions]`$_{Q}$ $\Rightarrow$ `[Descriptions]`$_{R}$
$\forall x[...x...]$	$\Rightarrow$	`All(x)[...x...]`
$f(x) \diamond y$	$\Rightarrow$	`x{f`$\diamond$`y}`
$f(x) \diamond f(y)$	$\Rightarrow$	`x{f`$\diamond$`f(y)}`
$x \diamond y$	$\Rightarrow$	(Undefined)

Table 5.10: From proposition to description.

Finally, descriptions of the same individual in the same scope

```
[x{A,B}, x{C,D}]
```

are merged together

```
[x{A,B,C,D}]
```

After a description is generated, it is processed such that empty descriptions (*i.e.* descriptions of individuals where there is nothing to verify) are removed, and identical attributes (*e.g.* `population=123` and `population=123`) are trimmed down to only one occurrence.

5.5 The data model

The data model is a list of object descriptions for every country and capital in the world. The data for this is acquired from WikiData by querying their service (see Appendix B) and the descriptions are translations from the resulting tables into object descriptions.

The object description for countries will have the name of the country, the country property, the capital, the continent, the population and the currency, *e.g.*

```
Sweden{country,capital=Stockholm,continent=Europe,
    population=10409248,currency=Swedish krona}
```

The capitals will have the name of the capital, the city property, the country they belong to and the population of the city, *e.g.*

```
Stockholm{city,country=Sweden,population=978770}
```

5.6 Verification

The verification is a comparison between descriptions generated from the propositions and the descriptions generated from the data model.

For the verification, we will be using the following notation:

Valid	✓	The description is fully valid.
Invalid	$\mathcal{X}$	The description is invalid.
Unknown	**?**	The validity can not be verified.

5.6.1 Verification of object descriptions

An interpreted object description holds if its properties (excluding constraints) is a comparable subset of the properties belonging to an object of the same name in the data model and if all constraints hold against the data model. It is invalid if a single property is invalid, and unknown if a single property can not have its validity determined.

Verification of properties

When two descriptions of the same name are compared, their properties are matched:

- Concepts: A concept holds if both descriptions share it.
- Attributes: An attribute holds if both descriptions share it and the values are comparable.

When two attributes are compared, we make a statement about if the attribute from the data model can justify the to-be-verified attribute. This in essence boils down to the semantics of the relations:

Attribute	Semantics
`a=x`	The attribute `a` is exactly `x` and nothing else.
`a>x`	The attribute `a` is something greater than `x`.
`a<x`	The attribute `a` is something less than `x`
`a`$\geq$`x`	The attribute `a` is something greater than `x` or is exactly `x`
`a`$\leq$`x`	The attribute `a` is something less than `x` or is exactly `x`
`a`$\approx$`x`	The attribute `a` is at most 30 % larger or 30 % smaller.

Table 5.11: The semantics of the binary relations.

The attribute holds only if the known value is the same as the claimed value or if it is for certain within the bound established by the relation.

Known value	Can verify
a=x	Any assertion that the value x belongs to.
a>x	Only an assertion with the $>$ relation and value x.
a<x	Only an assertion with the $<$ relation and value x.
a≥x	Only an assertion with the $\geq$ relation and value x.
a≤x	Only an assertion with the $\leq$ relation and value x.
a≈x	An assertion with the $\approx$ relation and value x or any assertion of which the upper and lower bound of the approximation holds for (*e.g.* an <y relation where the upper bound of ≈x is less than y).

Table 5.12: Assertion of the relations.

Note that values which are not enumerable (*i.e.* text) will not be verifiable using any relation other than the the identity relation.

Since there may be multiple attributes of the same name in the object description that is compared against, each has to be checked until one verify. If none verify, then it is invalid if one of the attributes can be determined to be invalid (*i.e.* strictly out of its bound). Otherwise it is unknown.

Verification of constraints

Verification of constraints is very similar to verification of attributes. The only difference is that the attributes have to be fetched first from the data model.

If an individual x has the attribute a constrained by the value of attribute a for an individual y using the < relation – then we first fetch the attribute a of individual x and attribute a of individual y from the data model, and compare that a(x) < a(y). If the constraint compare the same attributes (*e.g.* not comparing capital to continent) and these values compare as in the verification of the attributes, then the constraint holds. If there are multiple attributes of the same name to be fetched, then we try them all until one verify.

5.6.2 Verification of conditional descriptions

A conditional description is a description of the form:

$$[d_1, d_2, \ldots, d_n]_A \Rightarrow [d_1, d_2, \ldots, d_m]_B$$

A conditional description will hold if:

- One description out of the n descriptions in $[\ldots]_A$ is not valid.
- All descriptions in $[\ldots]_A$ are valid and all descriptions in $[\ldots]_B$ are valid.

5.6.3 Verification of universal descriptions

A universal description is a list of descriptions that must hold for every individual in the data model. Remember that a universal quantification is an expression of the form

$$\lambda x \mapsto \ldots x \ldots$$

which in turn will give a description of the form

$$\lambda x \mapsto [\ldots x\{\ldots\}\ldots]$$

In order to validate such a description, every individual in the data model has to be verified. Meaning that each description generated for each individual must be valid. A single invalid description makes the universal description invalid.

5.7 Multilingualism

The program above has been described primarily as verification of English texts. However, verification of texts in other languages can easily be achieved by extending the concrete grammars to those new languages. Since they are then based on the same abstract grammar, they will also then generate the same abstract syntax trees as the English texts with the same grammar, and thus be interpreted as the same propositions.

In this project there is a corresponding concrete grammar for Swedish, mostly just to showcase the point of how this could be achieved for more languages.

In order to not duplicate the data model by providing a Swedish alternative (which is however trivial with WikiData), the abstract syntax tree will be linearized in English instead when building the propositions.

6

A thorough example

To tie the project together we will study the path from text to verification using an example. For this, we will use the following English text:

```
The population of Sweden is about 10 million and its
capital is Stockholm. Sweden is the country with the
largest population.
```

Naturally, the first sentence is valid and the second sentence is invalid. To show the multilinguality of the interpretation, we will also study the Swedish grammatically equivalent text:

```
Sveriges befolkning är ungefär 10 miljoner och dess
huvudstad är Stockholm. Sverige är landet med den
största befolkningen.
```

6.1 Parsing the texts into abstract syntax trees

The first step of the verification is to transform the text into an abstract syntax tree. (Which is simply a matter of parsing with Grammatical Framework.) There is a risk that the parsing gives multiple abstract syntax trees – this happens when the parsed text is ambiguous in accordance with the grammar – however it is not of concern in this example.

The English text provides the following abstract syntax tree:

```
AddSentenceDoc (OneSentenceDoc (ConjSentence
(FactSentence (AttributeFact PopulationAttribute
(NameObject (MkName Sweden_CName)) (NumericValue
(AboutNumeric (IntMillionNumeric 10))))) (FactSentence
(AttributeFact CapitalAttribute (PronounObject (MkName
?12)) (NameValue (MkName Stockholm_CName))))))
(FactSentence (MaxObjectKindAttributeFact (NameObject
(MkName Sweden_CName)) CountryKind
PopulationAttribute))
```

Similarly, the Swedish text provides this abstract syntax tree:

```
AddSentenceDoc (OneSentenceDoc (ConjSentence
(FactSentence (AttributeFact PopulationAttribute
(NameObject (MkName Sweden_CName)) (NumericValue
(AboutNumeric (IntMillionNumeric 10))))) (FactSentence
(AttributeFact CapitalAttribute (PronounObject (MkName
?12)) (NameValue (MkName Stockholm_CName))))))
(FactSentence (MaxObjectKindAttributeFact (NameObject
(MkName Sweden_CName)) CountryKind
PopulationAttribute))
```

Observe that this tree is identical to that of the English abstract syntax tree (which is the case since they share the same abstract grammar and the texts are grammatically identical). From here on the English text and the Swedish text can be seen as the same.

6.2 Replacing the meta elements in the parsed tree

Before interpreting the parse tree, we have to replace all meta elements (which represent pronouns). Pay attention to the ?12 in the parse tree above, this one will be replaced with '<?>' in the tree below:

```
AddSentenceDoc (OneSentenceDoc (ConjSentence
(FactSentence (AttributeFact PopulationAttribute
(NameObject (MkName Sweden_CName)) (NumericValue
(AboutNumeric (IntMillionNumeric 10))))) (FactSentence
(AttributeFact CapitalAttribute (PronounObject (MkName
'<?>')) (NameValue (MkName Stockholm_CName))))))
(FactSentence (MaxObjectKindAttributeFact (NameObject
(MkName Sweden_CName)) CountryKind
PopulationAttribute))
```

6.3 Interpreting the parsed tree

With the abstract syntax tree ready for interpretation, it is just a matter of passing it to the recursive interpretation function, which gives the following proposition:

$$\texttt{(population(Sweden)} \approx \texttt{10000000} \wedge \texttt{capital(<?>) = Stockholm)} \wedge \forall x_0 \texttt{[(country(}x_0\texttt{)} \rightarrow \texttt{(population(}x_0\texttt{)} \leq \texttt{population(Sweden)} \wedge \texttt{country(Sweden)))]}$$

6.4 Naïve context-building

The last part of the interpretation is to solve the unknown <?> individual. In this example we get two contexts:

(1)

$$\begin{aligned}&(\texttt{population(Sweden)} \approx \texttt{10000000} \wedge \texttt{capital(Sweden)} = \\ &\texttt{Stockholm}) \wedge \forall \texttt{x}_0[(\texttt{country}(\texttt{x}_0) \rightarrow (\texttt{population}(\texttt{x}_0) \leq \\ &\qquad \texttt{population(Sweden)} \wedge \texttt{country(Sweden)}))]\end{aligned}$$

(2)

$$\begin{aligned}&(\texttt{population(Sweden)} \approx \texttt{10000000} \wedge \texttt{capital(10000000)} = \\ &\texttt{Stockholm}) \wedge \forall \texttt{x}_0[(\texttt{country}(\texttt{x}_0) \rightarrow (\texttt{population}(\texttt{x}_0) \leq \\ &\qquad \texttt{population(Sweden)} \wedge \texttt{country(Sweden)}))]\end{aligned}$$

This is because

$$\texttt{capital(<?>) = Stockholm}$$

can take the form of either

$$\texttt{capital(Sweden) = Stockholm}$$

and the more unreasonable

$$\texttt{capital(10000000) = Stockholm}$$

6.5 Building descriptions out of the propositions and verifying them

Finally, it is time to build descriptions. Given that we have two contexts, we will have to build descriptions twice.

6.5.1 Descriptions of the first context

In the first context we get the following descriptions:

$$\texttt{Sweden\{population}\approx\texttt{10000000,capital=Stockholm\}}$$

$$\begin{aligned}&\texttt{All(x}_0\texttt{)[[x}_0\texttt{\{country\}]}\Rightarrow \\ &\texttt{[x}_0\texttt{\{population}\leq\texttt{population(Sweden)\},Sweden\{country\}]]}\end{aligned}$$

Here the first object description about Sweden is valid ✓and the second universal description is invalid $\mathcal{X}$.

6.5.2 Descriptions of the second context

In the second context we get the following descriptions:

$$\texttt{Sweden\{population}\approx\texttt{10000000\}}$$

$$\texttt{10000000\{capital=Stockholm\}}$$

$$\texttt{All(x}_0\texttt{)[[x}_0\texttt{\{country\}]}\Rightarrow$$
$$\texttt{[x}_0\texttt{\{population}\leq\texttt{population(Sweden)\},Sweden\{country\}]]}$$

Here the first object description about `Sweden` is valid ✓, the second object description about `10000000` is unknown **?** (since there is no such individual in the data model), and the third universal description is invalid $\mathcal{X}$.

7

Evaluation

In order to verify the verification we can test the known truths in the system, *i.e.* that text generated from the data model holds and that the universal truths hold.

7.1 Sanity check against the data model

Our data model consists of a list of descriptions of countries, for example

```
Sweden{country,capital=Stockholm,continent=Europe,
    currency=Swedish krona,population=10409248}
```

and a list of descriptions of capitals, for example

```
Stockholm{city,country=Sweden,population=978770}
```

For each object description in the data model, we can generate a simple sentence corresponding to each attribute, for example

```
The capital of Sweden is Stockholm.
```

By parsing, interpreting and verifying these sentences, we can check that the program is sane, since each of them should be valid (which is a given, since the information in the text comes from the data source). Following are some examples:

Text		**Result**
`Sweden is a country.`	⇒	✓
`The capital of Sweden is Stockholm.`	⇒	✓
`The continent of Sweden is Europe.`	⇒	✓
`The currency of Sweden is Swedish krona.`	⇒	✓
`The population of Sweden is 10409248.`	⇒	✓

Table 7.1: Snapshot of the sanity check.

The sanity check can be summarized accordingly:

Count	Assertion
1468	properties are ✓
126	properties are **?**
0	properties are $\mathcal{X}$

Table 7.2: Summary of the sanity check.

Notably, the 126 properties which could not be validated are mostly a mismatch between the contents of the lexicon of common names in the grammar and the contents of the data model (*e.g.* the data model has a currency which is not present in the grammar).

7.2 Universal truths

The universal truths are truths about the world which we know should hold (in accordance to the data[1]). Notably, the following should be true:

```
People's Republic of China is the country with the
largest population.

Beijing is the city with the largest population.

Tuvalu is the country with the smallest population.

Yaren District is the city with the smallest
population.
```

By parsing, interpreting and verifying these sentences we get that each of them is valid without ambiguity ✓.

[1]The Vatican City is often counted as the smallest country, it is however not included in the data and the lexicon.

8

Discussion

Despite the project being quite successful in the end, there is an argument to be had that some choices taken throughout the project are questionable. It is also worth pondering over if the project is in a dead end or if there are improvements to be made which could advance the project further.

8.1 Encountered problems

Naturally some of the decisions taken throughout the project might in hindsight not be the best.

8.1.1 Properties of values and numbers as individuals

Consider the sentence

`The population of Sweden is about 10 million inhabitants.`

This would be interpreted as the formula

$$\texttt{population(Sweden)} \approx \texttt{10000000} \wedge \texttt{inhabitant(10000000)}$$

From this we can infer that

$$\texttt{inhabitant(10000000)}$$

But what are the exact semantics of this? We know from the text that it is the 'units' of the population, however this is not clear in the inference. This problem becomes even more apparent when a number might have multiple colliding properties, *e.g.*

$$\texttt{inhabitant(10000000)} \wedge \texttt{meter(10000000)}$$

A better formula which would solve this problem would perhaps be

$$\texttt{population(Sweden,inhabitant)} \approx \texttt{10000000}$$

since now the property of the value is encoded into the function of the relation.

This problem is derived from the decision of treating numbers as individuals, which has the benefit of making the interpretation of the grammar easier and works well under the circumstance that no properties of numbers are not needed.

8.1.2 Descriptions might do more harm than good

The decision to encode the formulas as descriptions might actually be harmful for the verification, notably in the sense that it only restructures the inferred information from the formulas, thus introducing an unnecessary middle-man.

As an example, compare the following formula

$$\forall \mathtt{x_0}[\mathtt{population(x_0)} \leq \mathtt{population(Sweden)}]$$

to its corresponding description

$$\mathtt{All(x_0)[x_0\{population} \leq \mathtt{population(Sweden)\}]}$$

Conceptually they both have the exact same functionality, so the actual benefits are seemingly minimal.

8.1.3 The verification is limited by the data model

The current iteration of the data model is very small (*i.e.* only countries and capitals, with only a few select properties) and not bidirectional (*e.g.* we can verify that Sweden has the currency Swedish krona, but not that the Swedish krona is a currency used by Sweden). If the grammar would be expanded to a larger domain, then so must also the data model, which could become a potential bottleneck.

The benefit of using WikiData in the first place is the access to the seemingly unbounded amount of available information, however this is not very well represented with the current data model.

8.2 Future work

Following, we will discuss some suggestions on how to continue this project.

8.2.1 Encoding properties better

The decision to interpret numbers as individuals, and properties of values as predicates, might not be the best approach. Instead, given a relation on the form

$$f(x) \diamond v$$

then any property of v should preferably be encoded into the relation by the function f during the interpretation.

8.2.2 Avoiding descriptions

Instead of transforming propositions into some intermediate representation, there is probably a lot of value to be gained from inferring all information from the propositions, and verifying each inference, thus avoiding the middle-man.

Alternatively, one could also instead use an established description logic or some ontology language instead of "hacking" together a data structure like the descriptions in this project.

8.2.3 Better integration with WikiData

Currently, WikiData is only used for querying a small subset of data, and use that data tabularly. This has the benefit of being quite simple, but the downside of being very restrictive.

It would be handy if this could be reversed, *i.e.* rather than compare the interpretations to queried data, one could maybe query data based on the interpretations. This would pose a much larger challenge, however would have the benefit of completely removing the bottleneck imposed by the data model.

8.2.4 Grammar library

Since the grammar currently is very limited, it would be preferable to extend it such that documents of different subjects and more languages could be interpreted. This could be achieved by building a library of grammars and defining interpretations for each of them.

Furthermore, it would be handy to have some application interface with interpretations for different categories such that a new grammar simply has to map their constructors to the functions in the interface.

8.2.5 Better feedback to the author

As of right now, all the program does is verifying the interpreted descriptions against the data. It would be preferable if the author of the text could get a more clear indication of which parts of the text that was unclear, ambiguous or incorrect rather than just a plain valid/invalid/unknown.

8.3 Related work

A lot of inspiration for this project has been acquired from the book 'Computational Semantics with Functional Programming' [1], that shows how to work with

interpretation of text using Montague grammar & semantics with Haskell.

Although a bit niche, text generation with Grammatical Framework has seen some research in the past. For instance this work with generating text for museum exhibits based on ontologies [14].

Using Montague grammar & semantics is also seemingly niche, however there is for instance an interesting project about inferring answers to questions about astronomy by using Montague grammar & semantics with functional programming [9].

Finally, this project was conducted in parallel with two projects about encyclopedia text generation using Grammatical Framework, a project studying the grammar of Wikipedia articles using Universal Dependencies, and finally a project developing a machine learning language model for Wikipedia articles.

9

Conclusion

In this project we have developed a program which can take a text corresponding to a Grammatical Framework grammar for geographical facts in English and Swedish, interpret it as first-order logic using the theory of Montague grammar & semantics, and verify it against data queried from WikiData.

Although the program is limited in scope, there is a lot of room for future improvements which could render a fully usable program for verification of encyclopedic text. The grammar can be improved by adding more ways of expressing geographical facts and be extended to cover more subjects in more languages and the interpretation can be improved to become more seamless.

Naturally, there is an argument to be had if this is the correct approach for verification of text. Nonetheless it will hopefully provide good insights for future development.

References

[1] Jan van Eijck and Christina Unger. *Computational semantics with functional programming*. 2010.

[2] Stefano Crespi Reghizzi, Luca Breveglieri, and Angelo Morzenti. *Formal Languages and Compilation*. 2019.

[3] Jon Barwise and Julius Moravcsik. *Selected papers of Richard Montague. Edited and with an introduction by Richmond H. Thomason*. 1982.

[4] David R. Dowty, Robert Eugene Wall, and Stanley Peters. *Introduction to Montague Semantics*. 1981.

[5] Alonzo Church. *An unsolvable problem in elementary number theory*. 1936.

[6] Michael Huth and Mark Ryan. *Logic in computer science: Modelling and reasoning about systems*. 2004.

[7] Alan Turing. *On Computable Numbers, with an Application to the Entscheidungsproblem*. 1937.

[8] Aarne Ranta. *Implementing Programming Languages*. 2012.

[9] Richard Frost and John Launchbury. *Constructing natural language interpreters in a lazy functional language*. 1989.

[10] Aarne Ranta. *Grammatical Framework*. 2004.

[11] Aarne Ranta. *Grammars as Software Libraries*. 2009.

[12] Krasimir Angelov. *The Mechanics of the Grammatical Framework*. 2011.

[13] Adrian Bielefeldt, Julius Gonsior, and Markus Krötzsch. *Practical Linked Data Access via SPARQL: The Case of Wikidata*. 2018.

[14] Dana Dannélls. *Generating Tailored Texts for Museum Exhibits*. 2008.

A

Converting Grammatical Framework grammar into Haskell data types

Fortunately, Grammatical Framework grammars can be exported as Haskell data types using the following Grammatical Framework command:

```
gf -make -output-format=haskell --haskell=lexical
   -lexical=[...] [{...}.gf]
```

With this command you provide the categories which are lexical (*i.e.* a list of words) to the `-lexical` parameter and the path to each top-level module of the abstract grammar for each language which you wish to include.

In this project we use the following:

```
gf -make -output-format=haskell --haskell=lexical
   -lexical=CName,CDName,Kind,Attribute
   CountriesEng.gf CountriesSwe.gf
```

This will give you the Haskell source code for the data types and a grammar file in the PGF format, which can with the Haskell library for PGF be included and interpreted as ordinary Haskell data types.

B

Queries for WikiData

In this project two queries has been used. The following for querying the countries of the world and relevant information:

```
SELECT DISTINCT ?countryLabel ?capitalLabel ?continentLabel
                ?currencyLabel ?population WHERE {
  ?country wdt:P31 wd:Q3624078 .
  ?country wdt:P36 ?capital    .
  ?country wdt:P30 ?continent  .
  ?country wdt:P38 ?currency   .

  # not a former country
  FILTER NOT EXISTS { ?country wdt:P31 wd:Q3024240 }

  # and no an ancient civilisation
  FILTER NOT EXISTS { ?country wdt:P31 wd:Q28171280 }

  # remove unused currencies
  FILTER NOT EXISTS { ?currency pq:P582 ?currencyEndDate }

  # get the current population
  ?country p:P1082 [ ps:P1082 ?population; pq:P585 ?date1 ]

  # remove duplicate entries for population
  FILTER NOT EXISTS {
    ?country p:P1082 [ pq:P585 ?date2 ]
    FILTER (?date2 > ?date1)
  }

  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "en"
  }
}
ORDER BY ?countryLabel
```

The following query for the capitals of the world and relevant information:

```
SELECT DISTINCT ?capitalLabel ?countryLabel ?population WHERE {
  ?country wdt:P31 wd:Q3624078 .
  ?country wdt:P36 ?capital    .

  # not a former country
  FILTER NOT EXISTS {?country wdt:P31 wd:Q3024240}

  # and no an ancient civilisation
  FILTER NOT EXISTS {?country wdt:P31 wd:Q28171280}

  # get the current population
  ?capital p:P1082 [ ps:P1082 ?population; pq:P585 ?date1 ]

  # remove duplicate entries for population
  FILTER NOT EXISTS {
    ?capital p:P1082 [ pq:P585 ?date2 ]
    FILTER (?date2 > ?date1)
  }

  SERVICE wikibase:label {
    bd:serviceParam wikibase:language "en"
  }
}
ORDER BY ?capitalLabel
```