MASTERS

A Coq module for natural logic

Huijben, R.H.

Award date:
2013

Link to publication
A Coq module for natural logic

Rutger Huijben

August 16, 2013
# Contents

1 Introduction ........................................ 4  
1.1 What is Coq? .................................... 4  
   1.1.1 A basic proof .............................. 5  
   1.1.2 Inductive Types ......................... 7  
1.2 Terminology .................................. 8  

2 Philosophical Background ......................... 9  

3 Formal Problem Statement ......................... 11  
3.1 Introduction .................................. 11  
3.2 Lambda Logical Forms ......................... 11  
   3.2.1 The Basics .............................. 11  
   3.2.2 Basic Types ............................ 12  
   3.2.3 Well formed LLFs ...................... 12  
   3.2.4 An Example ............................ 13  
   3.2.5 Syntax ................................ 14  
3.3 Lexicographical Knowledge .................... 14  
   3.3.1 Lexicographic Ordering ............... 15  
   3.3.2 Lexicographical Restrictions .......... 16  
3.4 Contexts ..................................... 17  
   3.4.1 What can be in a context .............. 17  
   3.4.2 Simple statements ..................... 18  
   3.4.3 Formalization .......................... 19  
   3.4.4 Monotonicity ........................... 20  
3.5 Truth .......................................... 22  
   3.5.1 Introduction ............................ 22  
   3.5.2 Basic Truth ............................. 22  
   3.5.3 Boolean Operators ...................... 23  
   3.5.4 Quantifiers ............................. 24  
   3.5.5 Beliefs ................................ 28  

4 Implementation ................................... 30  
4.1 Basic Types .................................. 30  
   4.1.1 e ....................................... 30  
   4.1.2 t ....................................... 31
Chapter 1

Introduction

Many formal logic systems originally have their roots in natural language, \( \neg \land \forall \) all have natural language counterparts, but when these languages became more rigidly defined and more powerful overtime they also drifted away from their natural language origins. As a result, most modern formal languages do not translate very well to natural language and back. In the paper *An Analytic Tableau System for Natural Logic*, by Reinhart Muskens[6], a solution to this problem is discussed. The authors propose to use a language they call Lambda Logical Forms (LLFs). LLFs are terms of a subset of typed lambda calculus that are designed to, on the one hand stay much closer to natural language while, on the other hand still be rigidly defined. This makes them an excellent tool for analysing and understanding everyday reasoning.

It will turn out that one level of representation, that of Logical Form, can meaningfully be identified with the language of an existing and well-understood logic, a restricted form of the theory of types. It is not difficult to devise inference systems for this language, and it is thus possible to study reasoning systems that are based directly on language.

(*An Analytic Tableau System for Natural Logic*, p. 1)

In this project we have implemented a module for this language in the proof assistant Coq. We have designed a model of the language, proven several lemmas that confirm useful properties of the language and created custom LTac tactics that give the user tools to prove true sentence within the subset of natural language that it represents.

1.1 What is Coq?

Coq is a interactive theorem prover, or proof assistant, a system that helps its users to write and verify correct proofs. Proof assistants are distinct from automated theorem provers in that they still require user input to complete a proof. Due to this, proof assistants are more
expressive than theorem provers, allowing the user to write more elaborate proofs for more complex settings. From the Coq website[3]:

Coq is a formal proof management system. It provides a formal language to write mathematical definitions, executable algorithms and theorems together with an environment for semi-interactive development of machine-checked proofs. Typical applications include the formalization of programming languages semantics (e.g. the CompCert compiler certification project or Java Card EAL7 certification in industrial context), the formalization of mathematics (e.g. the full formalization of the 4 color theorem or constructive mathematics at Nijmegen) and teaching.

There is already a theorem prover for LLFs in development by Lasha Abzianidze, but we feel that having a Coq implementation would still be very beneficial for a number of reasons.

Firstly, the theorem prover is limited in the kind of propositions it can verify, it takes as input two LLFs and tells you whether one can be derived from the other. Our implementation is more flexible, it simply implements the syntax of LLFs in a Coq module and provides the user with tools to write proofs for them. This means that while you can still derive inferences or prove that an inference does not hold, you are not limited to that one structure. Related to this, by implementing the model in Coq we are also able to prove properties of the model rather than just specific LLFs.

More importantly though, we believe that a great degree of automation in this process is not necessarily a good thing. Often, when you are interested in whether a sentence is true, you are also interested in why it is true. And while the prover does provide a proof tree for completed proofs, these trees are far less intuitively readable than a step by step proof for which you can see the current state of the proof after every step.

We have used CoqIde as our development environment for this project.

1.1.1 A basic proof

We have attempted to assume no advanced knowledge of the Coq system in this document and at least chapters 1-3 assume no knowledge of Coq at all, but we do feel it is useful to have some basic knowledge of how proving is done in Coq so we will use this section to walk through a small example proof in Coq. For a more comprehensive introduction to Coq we recommend the excellent Coq in a hurry[1] or the much more thorough Coq’art[2].

In this section we are going to prove the following basic tautology:

\[ \neg ((A \Rightarrow \neg B) \land (A \Rightarrow B) \land A). \]

The basic structure for proofs in Coq looks like this:
Lemma <LemmaName> : <The actual lemma that needs to be proven>.

Proof.
<Proof>
Qed.

A period marks the end of a line and each line is evaluated sequentially. The proof itself consists of a series of tactics that manipulate what is known as the proofstate. The proofstate is essentially a list of hypotheses that are assumed to be true and a list of goals that need to be proved. By evaluating each line before entering the next we can gradually build up the proof. If we enter our example in Coq the initial proofstate looks like this:

\((\neg A \rightarrow B) \land (A \rightarrow B) \land A\)

We have no hypotheses we can use, but we only have to prove our initial lemma. Now we can start applying tactics. We begin by using the \texttt{intro} tactic. This tactic can be used when the goal is an implication, it adds the right-hand side of the implication to the hypotheses, leaving the left-hand side as a goal. We can use it here because Coq internally represents a negation as an implication with false as the right-hand side (ie. \(\neg A\) is actually \(A \rightarrow \text{False}\)). Now the proofstate looks like this:

\begin{align*}
H: & (A \rightarrow \neg B) \land (A \rightarrow B) \land A \\
\text{False}
\end{align*}

We now have a conjunction as an hypothesis. It is typically preferable to split conjunctions in your hypotheses into their composite conjuncts as that allows you to refer to each of them separately. We can do this with the \texttt{destruct} tactic. After calling \texttt{destruct H} we have two hypotheses, one of which is again a conjunction, so we use \texttt{destruct} again. This leaves us with the following proofstate:

\begin{align*}
H: & A \rightarrow \neg B \\
H0: & A \rightarrow B \\
H1: & A \\
\text{False}
\end{align*}

The next tactic we are going to use is \texttt{apply}, this tactic takes a hypothesis that has the current goal as the right hand-side and replaces the goal with the left-hand side. After calling \texttt{apply H} (remember that negations are just implications with false as the right-hand side) we get the following proof state:

\begin{align*}
H: & A \rightarrow \neg B \\
H0: & A \rightarrow B \\
H1: & A \\
1)A \\
2)B
\end{align*}
We now have two goals, we first have to prove the current goal ($A$), but Coq reminds us that once we have done that we still have to prove a second goal ($B$). Since we already have $A$ as a hypothesis we can complete the current goal using \texttt{assumption}. Coq then automatically switches to the second goal. Here we will again use \texttt{apply}, but this time we will use it on $H1$ instead of on the goal. This is done by calling \texttt{apply $H0$ in $H1$} and it replaces $H1$ with $B$. Since we now again have our goal as a hypothesis we can complete the current goal, and with it the proof, by calling \texttt{assumption} again.

Of course, Coq has much more powerful tactics than the ones used in this toy example. We could have completed this entire proof in one step by using, for example, \texttt{intuition}.

1.1.2 Inductive Types

For this project we did not only have to complete proofs, our primary goal was to design a model for expressing LLFs. There are several ways to model things in Coq, but we mostly used inductive types. From Coq’Art:

> The inductive types of Coq extend the various notions of type definitions provided in conventional programming languages. They can be compared to the recursive type definitions found in most functional programming languages. However, mixing recursive types and dependant products makes the inductive type in Coq much more precise and expressive, up to the point where they can also be used to describe pure logic programming. (Coq’Art, p. 137)

Inductive types have a name, a type and a number of constructors. These constructors exhaustively define the ways that you can get an instance of the inductive type. As an example, this is a simple definition of a binary tree of natural numbers, implemented as an inductive type.

\[
\text{Inductive tree : Set :=}
\begin{align*}
\mid & \text{leaf : tree} \\
\mid & \text{node : tree -> nat -> tree -> tree.}
\end{align*}
\]

It has two constructors, one that states that a leaf is a tree and one that lets you combine two trees with a natural number to get another tree. Now if we have an instance of this type Coq provides us with a number of built in tactics to deal with it. Most prominently the \texttt{inversion} tactic, this tactic works on a hypothesis that is an instance of an inductive type, it looks up which constructors could have been used to build this instance and creates a new goal for each of those constructors. In this example, that could be used to do a proof by induction on the height of the tree (the leaf constructor is the base case and the node constructor is the induction hypothesis).
1.2 Terminology

Throughout this document we will use a number of conventions for notation. Coq code will be in teletype font, single quotes wrapped around operators indicate the infix version of this operator is used and single quotes around a natural language sentence indicate that this sentence is used in object language (e.g. examples).

Furthermore we will use specific variable names for certain types. As we will see, there are two basic types in the language, \( e \) and \( t \), the following variables will always be of the indicated type:

- \( o \) and \( o_1...o_n \) are of type \( e \).
- \( p, q, r \) and \( s \) are of type \( t \).
- \( w, x, y \) are of type \( e \to t \).
- \( z \) and \( z' \) are of type \( (e \to t) \to (e \to t) \to t \).
Chapter 2

Philosophical Background

Making sense of the structure of reasoning has always been an important goal of philosophy, going all the way back to Aristotle and the birth of western philosophy. The notion of tying this structure to a mathematical notation, however, doesn’t appear until Leibniz. Leibniz suggested that we should formalize our reasoning so that we could solve arguments with rigid, impartial mathematics rather than vague and often inconsistent arguments. This turned out to be a somewhat optimistic vision, but he did lay the foundations of modern formal logic and put forth the basic notion that language could be formalized. Much later, in the 19th century, Frege would build on these foundations with his Begriffsschrift [4], which was essentially the first incarnation of modern predicate logic, though his purpose for it was mathematical rather than philosophical. It was Russell who took up his ideas and started using logic to represent natural language.

Later yet, in the second half of the 20th century, Richard Montague developed his Montague Grammar. This was a mayor step forward in the process of formalizing natural language. For the first time, an attempt was made to devise a structural way of translating the entire foundations of natural language into a formal system with derivation rules [5] (rather than on a case by case basis such as almost all of Montagues predecessors had done). He was also the first to introduce Lambda Calculus to the field. Montague grammars marked a big change in the way people attempted to reconcile natural language with a formal logic system, but they were still far from perfect. Nowadays there are numerous philosophers, computer scientists and linguists working within this field. Some authors attempt to capture the subtleties of a single natural language concept, others attempt to create whole systems that provide a framework for embedding natural languages into a formal context.

A good example of the first approach is the paper Variable types for meaning assembly: a logical syntax for generic noun phrases introduced by most [8] by Christian Retoré, which explores a number of possibilities to improve upon current models of lexicographical knowledge, but does so by focussing on the subtleties that come from one specific ill defined quantifier (most). Retorés approach draws from the same history as the approach in the paper that we are using and as such they share many similarities (t and e as basic types, separating quanti-
A Coq module for natural logic

fier arguments into domain and predicate arguments, etc) but due to the much smaller scope he is able to use a far more involved model of lexicographical knowledge, introducing separate types for groups of things (i.e. Mars, the Sun and the Moon all have the type *celestial_body*) while also providing for ways in which those same constants can be of multiple types at the same time, depending on the context (Mars could also have the type *Snack*).

A good example of the second approach is the grammatical framework, introduced by Aarne Ranta in [7]. The grammatical framework allows a user to provide an abstract syntax that mirrors the syntactic structure of a natural language sentence. The key idea is that this framework is designed in a way that allows for automated translation from a sentence written using the abstract syntax to (theoretically) any natural language. This allows effortless translations between sentences in different languages provided that they have the same syntactic structure that fits into a defined abstract syntax.

Because most implementations in the grammatical framework (for obvious reasons) will attempt to stay as close as possible to the syntax of most natural language, a sentence in the grammatical framework will typically look similar to an LLFs, but the grammatical framework does not have a fixed syntax, instead requiring the user to define it himself. The grammatical framework needs the added flexibility that this provides because it is intended to be used in practical circumstances. A key difference between the grammatical framework and LLFs is that the grammatical framework is only interested in the semantic content of words in the sense that words from different languages need to translate to the same concept in the abstract syntax whereas LLFs are specifically designed to model the semantic of a sentence alongside its syntactic structure.

Due to this, while the overall approach of this project is falls somewhere between these two examples, it is somewhat closer to to Retoré’s approach than to the grammatical framework. LLFs are an attempt to provide a generic framework for embedding formal structure and meaning into natural language. There will often be subtleties that are not covered by the implementation as presented here, but the approach is fairly modular and can easily be extended to include more clever solutions for specific problems.
Chapter 3

Formal Problem Statement

3.1 Introduction

In this chapter we will give a formal specification of the system, which we will expand to a Coq module in chapter 4. We will provide this specification in terms of a series of axioms that the final module should adhere to.

3.2 Lambda Logical Forms

3.2.1 The Basics

From the paper:

For our purpose it will be of help to have representations of natural language expressions that are adequate both from a linguistic and from a logical point of view. At first blush, this may seem problematic, as it may be felt that linguistics and logic require completely different and competing properties from the representations they use, but in fact the typed lambda calculus provides what we need, or at least a good approximation to it.

(An Analytic Tableau System for Natural Logic, p. 2)

Lambda Logical Forms are defined as terms of a subset of typed lambda calculus (\( \lambda \rightarrow \)), specifically terms that are built up from variables and constants without logical semantic baggage (i.e. constants like =, \( \forall \), \( \exists \), \( \wedge \), \( \vee \), etc. cannot appear in a well formed LLF), though we will have constants that closely reflect the natural language counterparts of some of the more common logical operators.
What constants are allowed in an LLF is defined by its signature. A signature contains all the required information about the language that is being modelled by an LLF. It contains among other things the constants that can appear in an LLF and the basic lexicographical knowledge that comes with them, (e.g. the fact that all roses are flowers). We will use $\Sigma$ as a variable for signatures.

### 3.2.2 Basic Types

We will have two basic types, $t$ and $e$. Terms of type $e$ refer to the basic entities in the language, constants of type $e$ are names that designate the agents of the language. Terms of type $t$ are those terms that have an evaluable truth condition, usually these are sentences but they can also represent parts of other sentences that are evaluable separately (e.g. "A man walks" in the sentence "Arthur says that a man walks"). All LLF’s are terms of type $t$.

### 3.2.3 Well formed LLFs

We will first define what qualifies as a well formed LLF, these are just the usual rules for typed lambda calculus.

**Definition 1.**

A well formed LLF is any term that can be derived as having type $t$ using the following axioms:

- **Constants**
  
  \[
  \frac{c : \sigma \in \Sigma}{\Sigma; \Gamma \vdash c : \sigma}
  \]

- **Variables**
  
  \[
  \frac{x : \sigma \in \Gamma}{\Sigma; \Gamma \vdash x : \sigma}
  \]

- **Application**
  
  \[
  \frac{\Sigma; \Gamma \vdash M : \sigma \rightarrow \tau \quad \Sigma; \Gamma \vdash N : \sigma}{\Sigma; \Gamma \vdash MN : \tau}
  \]
Abstraction

\[ \Sigma; \Gamma; \alpha : \sigma \vdash M : \tau \] \[ \Sigma; \Gamma \vdash \lambda \alpha : \sigma. M : \sigma \to \tau \]

3.2.4 An Example

For a concrete example, suppose we want to translate the English sentence ‘The woman who killed Bianca Loves Clark’. We first have to find the types of the constants involved. Bianca and Clark are names so they are of type \( e \). Nouns and verbs of arity one (we will refer to the number of agents involved in an verb as its arity) have to be of type \( e \to t \) because sentences like ‘Diana walks’ or ‘Edward is a man’ have to be of type \( t \). Likewise, verbs of arity two are of type \( e \to e \to t \). Generally, terms of type \( e \to t \) function like properties of an entity, thus the term \( \text{Man} \) represents the property of being a man and \( \lambda x. (\text{Killed} \; x \; \text{Fiona}) \) represents the property of having killed Fiona.

We will consider words like ‘The’, ‘A’ and ‘Each’ to represent quantifiers. Each quantifier takes two arguments of type \( e \to t \), the first is a restriction of the kind of entities that is being quantified over, the second is the property that should hold for the quantifier to become true, thus "Each bird flies" translates to \( \forall o. \text{Bird} \; o \; \text{flies} \). Consequently the type of these quantifier words is \((e \to t) \to (e \to t) \to t \). Using quantifier words in this way leads to a somewhat unintuitive order of evaluation for some sentences. When parsing the sentence ‘The man who stands is brave’ many people would consider \( \text{Is} \) to be the top level operator and \( \text{The} \) to be applied to \( \text{Man} \) on the lowest level, instead, \( \text{The} \) is the top level operator and \( \text{Is} \) is syntactic sugar that we will not even translate. As we will see in section 3.4.4, this interpretation makes things a lot simpler when we start to evaluate monotonicity properties.

That leaves the word ‘Who’, we will consider the function of who to be to connect two terms of type \( e \to t \), where the resulting term represents the conjunction of the two subterms. Thus \( \text{Who Man Walk} \) represents the property of being a man who walks.

To return to our example, we now have \( \Sigma = \{ \text{The} : (e \to t) \to (e \to t) \to t, \text{Woman} : e \to t, \text{Who} : (e \to t) \to (e \to t) \to e \to t, \text{Kill}, \text{Love} : e \to e \to t, \text{Bianca}, \text{Clark} : e \} \). With this we can form the LLF \( \text{The (Who Woman (\lambda x (\text{Kill} \; x \; \text{Bianca})) (\lambda y (\text{Love} \; y \; \text{Clark})))} \) in the following manner:

\[
\begin{array}{c}
\Sigma, y : e \vdash \text{Love} : e \to e \to t \\
\Sigma, y : e \vdash \text{Love} : e \to e \to t \\
\Sigma, y : e \vdash \text{Love} : e \to e \to t \\
\Sigma, y : e \vdash \text{Love} : e \to e \to t \\
\Sigma, y : e \vdash \text{Love} : e \to e \to t \\
\Sigma, y : e \vdash \text{Who Woman (\lambda x (\text{Kill} \; x \; \text{Bianca})) (\lambda y (\text{Love} \; y \; \text{Clark})))} : t \\
\end{array}
\]
Where the subproof A is as follows:

\[
A \\
\Sigma \vdash \text{Who} : (e \to t) \to (e \to t) \to e \to t \\
\Sigma \vdash \text{Woman} : e \to t \\
\Sigma, x : e \vdash \text{Kill} : e \to e \to t \\
\Sigma, x : e \vdash \text{Kill}_x : e \to t \\
\Sigma, x : e \vdash \text{Bianca} : e \\
\Sigma \vdash \lambda x \text{(Kill}_x \text{Bianca)} : e \to t
\]

3.2.5 Syntax

As stated above, one of the pleasant properties of LLFs is that they remain semantically very close to their corresponding natural language sentences. However, the LLF we achieved in the above example still seems very puzzling at first glance. There are two major problems that cause this.

Firstly the verbs are not conjugated, the sentences ‘The man who loves Fiona loved Hannah’ and ‘The man who loved Fiona loves Hannah’ have clearly different meanings, but they translate to the same LLF (\(\text{The} \ (\text{Who Woman} \ (\lambda x (\text{Love} x \text{Fiona})) \ (\lambda y (\text{Love} y \text{Hannah})))\)). This is a very interesting problem that unfortunately falls outside of the scope of this project, we will simply assume that the order of events can be derived from background knowledge and the context in which the sentence is uttered.

The second problem is that the order of the words is very awkward, the sentence looks like someone took a normal sentence and randomly shuffled some words around. The cause of this is that most languages use infix notations for their binary operators, switching to a prefix style notation increases the distance between operators and their operands and makes it difficult for us to parse the sentence. Luckily their is an easy fix for this, if we simply switch to an infix notation for the words \text{Who}, \text{Kill} and \text{Love} the sentence becomes \text{The} (\text{Woman} ‘\text{Who}’ (\lambda x (x ‘\text{Kill}’ \text{Bianca})) (\lambda y (y ‘\text{Love}’ \text{Clark})) which is much closer to the original natural language sentence. We will use ‘ ’ around an operator to indicate that we are using an infix version of that operator.

3.3 Lexicographical Knowledge

Humans have access to an enormous pool of background knowledge that informs them of the relations between words: we know that all swans are birds, that if every duck can swim then most ducks can swim and that something can’t be both a cat and a dog. We will refer to this kind of background information as lexicographical knowledge. We will pay particular attention to two instances of lexicographical knowledge.
3.3.1 Lexicographic Ordering

First are those cases where we know some set of entities to be a superset of another set of entities (the birds and the swans mentioned above). This is modelled as a number of trees of constants that among them contain all basic properties (constants of type $e \to t$) and quantifiers (constants of type $(e \to t) \to (e \to t) \to t$) in the language. The tree represents a lexicographical ordering where every entity that has the child property also has the property property. We will denote the fact that a property $x$ is an ancestor of a property $x'$ as $x' \subseteq x$ or $x'$ is a subset of $x$.

We also use this relation to model the fact that we use Who as a conjunction between two properties, a man who is walking is still a man, hence $(\text{Man 'Who' Walk}) \subseteq \text{Man}$.

**Definition 2.**
Formally the *subset relation* is defined by the following axioms:

**Direct Parents**

\[
\frac{x \subseteq y \in \Sigma}{\Sigma \vdash x \subseteq y}
\]

**Transitive Ordering**

\[
\frac{\Sigma \vdash x \subseteq y \quad \Sigma \vdash y \subseteq z}{\Sigma \vdash x \subseteq z}
\]

**Reflexive Ordering**

\[
\Sigma \vdash x \subseteq x
\]

**Who Ordering 1**

\[
(x \ 'Who' \ y) \subseteq x \quad \text{with } \Sigma \vdash x, y : e \to t
\]

**Who Ordering 2**

\[
(x \ 'Who' \ y) \subseteq y \quad \text{with } \Sigma \vdash x, y : e \to t
\]
The trees that correspond to the subset of natural language that we are interested in are as follows:\(^1\):

### Quantifier
- Most
- Few
- Some
- Many
- Each
- No
- A
- The

### Binary Verb
- Like
- Hurt
- Touch
- Move
- Sleep
- Run
- Walk
- Fly
- Dream
- Snore

### Unary Verb
- Move
- Sleep
- Touch

### Creature
- Person
  - Human
  - Student
  - Man
  - Woman
  - Bachelor
- Dog
  - Hound
- Bird
  - Penguin
  - Crow
  - Lark

### 3.3.2 Lexicographical Restrictions

The second kind of lexicographical knowledge that we will examine are lexicographical restrictions. A lexicographical restriction is the knowledge that the same entity can’t have two properties simultaneously, no entity can be simultaneously blue and red all over so there is a lexicographical restriction between the property of being blue all over and the property of being red all over. We will refer to this as **Blue All Over is incompatible with Red All Over**, denoted `Blue All Over \( \not\approx \) Red All Over`.

**Definition 3.**

The following axioms define the *incompatibility relation*:

**Base Restrictions**

\[
\frac{x \not= y \in \Sigma}{\Sigma \vdash x \not= y}
\]

\(^1\)Note that **Some** and **A** are actually synonymous, the order they are placed in the tree here is arbitrary, and since they are both well defined quantifiers it is perfectly possible to derive **Some** from **A** using its definition.
Symmetric Restrictions

\[ \Sigma \vdash x \not\approx y \quad \text{with} \quad \Sigma \vdash x, y : e \rightarrow t \]

Restriction Ordering

\[ \Sigma \vdash x' \subseteq x \quad \Sigma \vdash x \not\approx y \quad \text{with} \quad \Sigma \vdash x, y : e \rightarrow t \]

These are the basic restrictions that are present in the subset of natural language that we are modelling:

- Man \not\approx Woman
- Lark \not\approx Crow
- Lark \not\approx Penguin
- Crow \not\approx Penguin
- Dog \not\approx Person
- Dog \not\approx Bird
- Bird \not\approx Person

3.4 Contexts

It should be obvious that the truth of an LLF depends on the environment in which it is evaluated, to evaluate the sentence ‘Alice is working’ you have to look at the world and observe whether Alice is currently working. In order to model this we will use the concept of a context. We will define a context as a set of terms of type \( t \) that together provide an accurate description of the current state of the world.

3.4.1 What can be in a context

We want a context to be complete, that is, for every \( p \in c \) if \( p \Rightarrow q \) then \( q \in c \). Doing this naively would mean that every consistent context has infinite length. This is a problem because we would like to be able to enumerate contexts so we can use them as counterexamples in proofs. To remedy this we will only allow certain kinds of statements in a context: simple statements, who statements and ill defined quantifiers. Together, these statements can provide an accurate description of the world and all other statements that are true in a given context can be derived from them. We will go over each category briefly.
3.4.2 Simple statements

A simple statement is any statement that is either a belief statement or consist of one or more constants of type $e$ applied to a unary verb, a binary verb or a noun. These are the basic facts of the world: ‘John Walks’, ‘Sandra is a Woman’, ‘Marvin likes Jenny’ and ‘Lara believes the sky is blue’. They provide the information that defines whether other statements are true.

Definition 4.
Formally we will say that an LLF $p$ is simple under signature $\Sigma$ if $p \in S$ where $S$ is defined as follows:

$U$ is the set of all LLFs $x \circ o$ where $x$ is a constant of type $e \rightarrow t$ and $o$ is a constant of type $e$ in $\Sigma$.

$Bi$ is the set of all LLFs $x \circ_1 o_2$ where $x$ is a constant of type $e \rightarrow t \rightarrow t$ and $o$ is a constant of type $e$ in $\Sigma$.

$Be$ is the set of all LLF $b \circ o q$ where $b \in \{Belief, Say\}$, $o$ is a constant of type $e$ and $q$ is of type $t$ in $\Sigma$.

$S ::= U \cup Bi \cup Be$

Who Statements

Who statements are statements like ‘Michael is a student who likes beer’. Essentially these statements are a conjunction of two simple statements, ‘Michael is a student’ and ‘Michael likes beer’ in this example. Who statements can always be trivially derived from simple statements and as such it might be a little unintuitive that they are allowed in a context, but a number of practical considerations made it convenient to have access to who statements in a context, we will get back to this in section 3.5.

Definition 5.
Formally we will say that an LLF $p$ is a who statement under signature $\Sigma$ if and only if $p = (Who \ x \ y) \circ o$ where $x$ and $y$ are of type $e \rightarrow t$ and $o$ is a constant of type $e$ in $\Sigma$.

Ill Defined Quantifiers

For the final kind of statement that we will allow in a context we have to make a distinction between well defined quantifiers and ill defined quantifiers. Well defined quantifiers are those quantifiers for which we have an exhausting definition, for instance $\Lambda$ functions exactly like the existential quantifier, for these quantifiers we will give definitions in section 3.5.4. The
quantifiers that we are interested in for this section are those for which we have no exhaustive
definitions, words like many and few. For these quantifiers we have no way of deriving them
from simple statements (no matter how many statements like 'o is a woman who walks' are
in a given context we can never derive 'Many women walk' from them) so they have to be
part of the context.

Definition 6.
We will say that an LLF \( p \) is an ill defined quantifier statement under signature \( \Sigma \) if \( p = z \ x \ y \),
where \( x \) and \( y \) are of type \( e \rightarrow t \) in \( \Sigma \) and \( z \in I \) where \( I \) is the set of ill defined quantifiers in \( \Sigma \).

For the subset of natural language that we are modelling \( I = \{ \text{Many}, \text{Most}, \text{Few} \} \).

3.4.3 Formalization

Given all of that, we can now define the axioms concerning the consistency of a context.
Informally, a context is consistent if the world it models contains no contradictions and only
contains statements of the above three kinds. In the remainder of the document we will only
talk about consistent contexts, we are not interested in what happens to the truth value of
LLF under inconsistent contexts (in the Coq implementation they are neither false nor true).

Definition 7.
Formally a context \( c \) is consistent if the following rule as well as the rules in definition 8 hold:

**Context Classification**

\[
\frac{p \in c \quad \Sigma \vdash p : t}{\text{Cont}(p)}
\]

Where \( \text{Cont}(p) ::= p \) is Simple | \( p \) is a who statement | \( p \) is an ill defined quantifier statement

**Who Derivation in Contexts**

\[
\frac{x \ o \in c \quad y \ o \in c}{(x \ 'Who' \ y) \ o \in c} \quad \text{with } \Sigma \vdash o : e, x, y : e \rightarrow t
\]

\[
\frac{(x \ 'Who' \ y) \ o \in c}{x \ o \in c} \quad \text{with } \Sigma \vdash o : e, x, y : e \rightarrow t
\]

\[
\frac{(x \ 'Who' \ y) \ o \in c}{y \ o \in c} \quad \text{with } \Sigma \vdash o : e, x, y : e \rightarrow t
\]

**Lexicographic Ordering in Contexts**
\[ \Sigma \vdash x \subseteq y \quad x \in e \quad \text{with} \quad \Sigma \vdash o : e, x, y : e \rightarrow t \]

\[ \Sigma \vdash x \not\approx y \quad x \in e \quad \text{with} \quad \Sigma \vdash o : e, x, y : e \rightarrow t \]

**Lexicographic Restrictions in Contexts**

3.4.4 Monotonicity

Of special interest to this project are the so called monotonicity rules. These transformation rules are given little interest in most formal logic systems, but they play a large role in how humans reason in everyday life. It is immediately obvious to us that the sentence ‘Many people walk, therefore many people move’ is true, yet proving that this is true requires a concept of lexicographic knowledge that most formal logic systems do not provide.

Monotonicity defines one of the ways we can use the lexicographic ordering to deduce new truths. For example: we know that all crows are birds, therefore if we know that ‘A crow flies’ is true then we also know that ‘A bird flies’ is true. This derivation only works if the first argument of the original LLF (crow) is a descendant of the first argument of the goal LLF (bird) in the lexicographical ordering. We will call this upward monotonicity and because it applies to the first argument of A we say that A is upwards monotonic in its first argument.

The trick works in reverse for Each, if we know that ‘Each bird flies’ is true then we also know that ‘Each crow flies’ is true. This is because Each is downward monotonic in its first argument.

More general, we say that a quantifier is upwards monotonic in an argument if, given an LLF with that quantifier as its outermost operator, you can replace that argument with a property that is higher up the lexicographic ordering without invalidating the truth of the LLF. Similarly, a quantifier is downward monotonic in an argument if you can instead replace that argument with a property that is lower up the lexicographic ordering.

Each quantifier can have monotonicity in either direction for both its arguments. We will denote direction with an upward or downward arrow in superscript and the argument to which it applies in subscript, so \( z_1^\uparrow \) means that \( z \) is upwards monotonic in the first argument.

Note that the rules that we will give for the well defined quantifiers already obey monotonicity so we only have to define it for the ill defined quantifiers.

The following is a table of each quantifier in the subset of natural language that we are looking at with the directions of their monotonicity.
Definition 8.
The following rules extend the definition of a consistent context with quantifier monotonicity:

**Upwards Monotonicity 1**

\[
\begin{align*}
& z \ x \ y \in c \\
& \Sigma \vdash z^1 \\
& \Sigma \vdash x \subseteq x' \\
\end{align*}
\]

\[
\begin{align*}
& \Sigma^t \ y; x \to t; z \to t \\
& \Sigma^t \ x, y; x \to t; z \to t \\
\end{align*}
\]

**Upwards Monotonicity 2**

\[
\begin{align*}
& z \ x \ y \in c \\
& \Sigma \vdash z^1 \\
& \Sigma \vdash y \subseteq y' \\
\end{align*}
\]

\[
\begin{align*}
& \Sigma^t \ y, y'; x \to t; z \to t \\
& \Sigma^t \ x, y, y'; x \to t; z \to t \\
\end{align*}
\]

**Downwards Monotonicity 1**

\[
\begin{align*}
& z \ x' \ y \in c \\
& \Sigma \vdash z^1 \\
& \Sigma \vdash x \subseteq x' \\
\end{align*}
\]

\[
\begin{align*}
& \Sigma^t \ y; x \to t; z \to t \\
& \Sigma^t \ x, y; x \to t; z \to t \\
\end{align*}
\]

**Downwards Monotonicity 2**

\[
\begin{align*}
& z \ x' \ y \in c \\
& \Sigma \vdash z^1 \\
& \Sigma \vdash y \subseteq y' \\
\end{align*}
\]

\[
\begin{align*}
& \Sigma^t \ y, y'; x \to t; z \to t \\
& \Sigma^t \ x, y, y'; x \to t; z \to t \\
\end{align*}
\]

**Quantifier Monotonicity**

\[
\begin{align*}
& z \ x \ y \in c \\
& \Sigma \vdash z \subseteq z' \\
\end{align*}
\]

\[
\begin{align*}
& \Sigma^t \ y; x \to t; z \to t \\
& \Sigma^t \ x, y; x \to t; z \to t \\
\end{align*}
\]
3.5 Truth

3.5.1 Introduction

We will now go into the definition of the $\models$ relation, this relation defines the truth of an LLF given a context. It is the core focus of the project and by far the largest hurdle that we had to overcome came from how we should define it. The biggest problem we faced shows itself clearest in defining the operator $\text{Not}$. The first definition that springs to mind is something like:

$$\text{Not Derivation}$$

$$\frac{\neg(c \models_\Sigma p)}{c \models_\Sigma (\text{Not } p)} \text{ with } \Sigma \vdash p : t$$

The problem is that Coq does not allow the negation of a inductive type to appear in the constructors for that type. The reason for this is that in order to evaluate $\neg(c \models_\Sigma p)$ you would have to evaluate all the constructors for $\models$ and find no way to construct $c \models_\Sigma p$, one of those constructors being the above definition. This is not really a problem for this specific case but allowing it in general can lead to logical inconsistencies so Coq returns an error if you attempt it.

We cannot really drop the concept of negation from the language altogether though, so instead we will model the truth-making relation as two separate functions: $\models$ and $\not\models$. $c \models_\Sigma p$, pronounced $p$ is true in $c$, denotes that $p$ can be derived from context $c$ given signature $\Sigma$. $c \not\models_\Sigma p$, pronounced $p$ is false in $c$, on the other hand models that $\neg p$ can be derived from $c$ given $\Sigma$ (We will sometimes drop the $\Sigma$ if it is clear which signature is being used).

This solution neatly sidesteps the problem by allowing us to define both when something is false and when something is true, but it doubles the number of constructors that we need to define. Most of the definitions for $\not\models$ are fairly obvious negations of there positive counterparts though, so the actual increase in workload was not quite that bad. Though proving that the usual properties of true and false (e.g. excluded middle) still hold proved to be quite a challenge, as we will see in section 5.2.

3.5.2 Basic Truth

We will start with a very basic definition of both relations and then extend this definition over the rest of this chapter by adding rules for each constant that can produce a term of type $t$.

Definition 9.
The following rules provide a basic definition for the $\text{is true in}$ relation and the $\text{is false in}$ relations. This definition is extended in definitions 10, 11, 12, 13, 14, 15 and 16.
A Coq module for natural logic

Basic Truth Derivation

\[ p \in c \quad \text{with } \Sigma \vdash p : t \]

Basic False Derivation

\[ \text{Cont}(p) \quad p \notin c \quad \text{with } \Sigma \vdash p : t \]

3.5.3 Boolean Operators

Next we extend this definition with the basic boolean operators, there are a lot of subtleties to the use of these concepts in natural language, but this falls outside the scope this project. We will use the common definition used in propositional logic.

Definition 10.
These rules extend definition 9 with the definition for boolean operators.

And Derivation

\[ \begin{align*}
& c \models \Sigma p \\
& c \models \Sigma q \\
\hline
& c \models \Sigma (p \text{ 'And' } q) \quad \text{with } \Sigma \vdash p, q : t
\end{align*} \]

\[ \begin{align*}
& c \not\models \Sigma p \\
\hline
& c \not\models \Sigma (p \text{ 'And' } q) \quad \text{with } \Sigma \vdash p, q : t
\end{align*} \]

\[ \begin{align*}
& c \not\models \Sigma q \\
\hline
& c \not\models \Sigma (p \text{ 'And' } q) \quad \text{with } \Sigma \vdash p, q : t
\end{align*} \]

Or Derivation

\[ \begin{align*}
& c \models \Sigma p \\
\hline
& c \models \Sigma (p \text{ 'Or' } q) \quad \text{with } \Sigma \vdash p, q : t
\end{align*} \]

\[ \begin{align*}
& c \models \Sigma q \\
\hline
& c \models \Sigma (p \text{ 'Or' } q) \quad \text{with } \Sigma \vdash p, q : t
\end{align*} \]

\[ \begin{align*}
& c \not\models \Sigma p \\
& c \not\models \Sigma q \\
\hline
& c \not\models \Sigma (p \text{ 'Or' } q) \quad \text{with } \Sigma \vdash p, q : t
\end{align*} \]
A Coq module for natural logic

Not Derivation

\[
\frac{(c \not\models \Sigma p)}{c \models \Sigma (\text{Not } \Sigma p) \quad \text{with } \Sigma \vdash p : t}
\]

\[
\frac{(c \models \Sigma p)}{c \not\models \Sigma (\text{Not } \Sigma p) \quad \text{with } \Sigma \vdash p : t}
\]

Implies Derivation

\[
\frac{c \models \Sigma q}{c \models \Sigma (p \ '\text{Implies'} \ q) \quad \text{with } \Sigma \vdash p, q : t}
\]

\[
\frac{c \models \Sigma \text{Not } p}{c \models \Sigma (p \ '\text{Implies'} \ q) \quad \text{with } \Sigma \vdash p, q : t}
\]

\[
\frac{c \models \Sigma p \quad c \not\models \Sigma q}{c \not\models \Sigma (p \ '\text{Implies'} \ q) \quad \text{with } \Sigma \vdash p, q : t}
\]

3.5.4 Quantifiers

Next we have the quantifier words, these definitions may not always be self evident, so we will go over them in some more detail.

\(\mathbf{A}\)

We will consider \(\mathbf{A}\) as it is used in the sentence ‘A man is walking his dog’ where it is semantically equivalent to the existential quantifier. As with all quantifier words \(\mathbf{A}\) takes two terms of type \(e \rightarrow t\) to produce a term of type \(t\). Intuitively the first of these properties represents the domain that is being quantified over and the second represents the property that needs to be satisfied. Since both arguments constrain the domain in the same manner they can be swapped without changing the meaning of the sentence, \(\mathbf{A} \mbox{ Woman Walk and A Walk Woman}\)\(^2\)

\(^2\)The second example looks awkward because the English grammar doesn’t allow verbs to take up that grammatical position. There is a natural language sentence that corresponds to this LLF, though, it is something like ‘An entity that walks is a woman’ or, since that is ambiguous, ‘At least one entity that walks is a woman’.

/department of computer science
have the same truth condition.

Translating this meaning into axioms is fairly straightforward, we have an axiom for the positive version which states that \( A \ x \ y \) is true if there is an \( o \) for which \( x \ o \) and \( y \ o \) both hold and an axiom for the negative version which states that \( A \ x \ y \) is false if for every \( o \) either \( x \ o \) or \( y \ o \) fails to hold.

**Definition 11.**
The following rules extend definition 9 with a definition for \( A \).

\[
\begin{align*}
\text{A Derivation} & \\
\frac{c \models_\Sigma x \ o \quad c \models_\Sigma y \ o}{c \models_\Sigma A \ x \ y} \quad \text{with } \Sigma \vdash o : e, x, y : e \to t \\
& \\
\frac{\forall o,e \ (c \not\models_\Sigma ((x \ o \ \text{'And'} y \ o)))}{c \not\models_\Sigma A \ x \ y} \quad \text{with } \Sigma \vdash x, y : e \to t
\end{align*}
\]

**Some**

A number of test cases from the Prolog test suite (see section 5.3) use the word **Some**. The semantic content of **Some** is identical to \( A \), thus they will be considered to be synonymous.

**Definition 12.**
The following rules extend definition 9 with a definition for **Some**.

\[
\begin{align*}
\text{Some Derivation} & \\
\frac{c \models_\Sigma \text{Some} \ x \ y}{c \models_\Sigma A \ x \ y} \quad \text{with } \Sigma \vdash x, y : e \to t \\
& \\
\frac{c \not\models_\Sigma \text{Some} \ x \ y}{c \not\models_\Sigma A \ x \ y} \quad \text{with } \Sigma \vdash x, y : e \to t
\end{align*}
\]
Each

If $A$ is the natural language equivalent of the existential quantifier then $\text{Each}$ is the equivalent of the universal quantifier. Unlike with $A$ the arguments of $\text{Each}$ do serve a different purpose, the first argument indicates the domain of the quantifier while the second indicates the condition that has to be satisfied. Thus $\text{Each Crow Bird}$ and $\text{Each Bird Crow}$ (in natural language ‘Each crow is a bird’ and ‘Each bird is a crow’ respectively) have different meanings, the first is a tautology, the second is only true if there are no non-crow birds.

Again the axiomatization is straightforward. On the positive side, $\text{Each } x y$ is true if for every $o$, $x o$ is only true if $y o$ is also true. On the negative side, if there is an $o$ for which $x o$ is true and $y o$ is false then $\text{Each } x y$ is also false.

**Definition 13.**
The following rules extend definition 9 with a definition for $\text{Each}$.

$$
\forall_{o,e} (c \models_{\Sigma} (x o \text{ 'Implies' } y o)) \quad \text{with } \Sigma \vdash x, y : e \rightarrow t
$$

$$
\exists_{o,e} (c \models_{\Sigma} (x o \text{ 'And' (Not } (y o)))) \quad \text{with } \Sigma \vdash x, y : e \rightarrow t
$$

The

The word $\text{The}$ is slightly trickier because its meaning in natural language is somewhat vague. We will use the common interpretation that $\text{The}$ is the unique existential quantifier, thus the sentence ‘The king of France is bald’ is only true if there is exactly one king of France and he is bald. There are a number of cases in which this interpretation is a bit shaky, the sentence ‘The guy that works in the museum called’, for example, can probably still be considered true if the museum has more than one male employee (or if there is more than one museum for that matter), but this can usually be hand-waved by restricting the context to only the relevant facts (e.g. there may be many men working for the museum, but the speaker only knows one of them and the listener is aware of this).

The axiomatization of the positive side uses a straightforward implementation of the definition of the unique existential quantifier. The negative side is split into two cases, $\text{The } x y$ is false if either there is more than one entity for which $x o$ and $y o$ are true or there are none.
Definition 14.
The following rules extend definition 9 with a definition for The.

The Derivation

\[
\frac{c \models_{\Sigma} (x \ o \ 'And' \ y \ o)}{c \models_{\Sigma} \text{The } x \ y} \quad \forall o':e \ (c \models_{\Sigma} (x \ o' \ 'And' \ y \ o') \implies (o' = o)) \quad \text{with } \Sigma \vdash o : e, x, y : e \to t
\]

\[
\frac{c \models_{\Sigma} \text{Not } A \ x \ y}{c \not\models_{\Sigma} \text{The } x \ y} \quad \text{with } \Sigma \vdash x, y : e \to t
\]

\[
\frac{c \models_{\Sigma} x \ o \ 'And' \ y \ o \quad c \models_{\Sigma} x \ o' \ 'And' \ y \ o' \quad o \neq o'}{c \not\models_{\Sigma} \text{The } x \ y} \quad \text{with } \Sigma \vdash o, o' : e, x, y : e \to t
\]

No

No is merely the inverse of A. If ‘A duck flies’ is true then ‘No duck flies’ is false and vice versa. No can thus be implemented solely in terms of A and Not.

Definition 15.
The following rules extend definition 9 with a definition for No.

No Derivation

\[
\frac{c \models_{\Sigma} \text{Not } (A \ x \ y)}{c \models_{\Sigma} \text{No } x \ y} \quad \text{with } \Sigma \vdash x, y : e \to t
\]

\[
\frac{c \models_{\Sigma} A \ x \ y}{c \not\models_{\Sigma} \text{No } x \ y} \quad \text{with } \Sigma \vdash x, y : e \to t
\]

Ill Defined Quantifiers

There are three quantifiers (in the subset of natural language we are looking at) that are not in the above list: Few, Many and Most. These quantifiers are a special case, as they are ill defined and can only be derived from the context through monotonicity (see section 3.4.4). Thus these quantifiers do not have special axioms dedicated to them.
3.5.5 Beliefs

The final category we will be looking at are axioms that revolve around two core concepts, beliefs and claims. Claim statements are sentences like 'Karen says Daniel is the killer' and belief statements are sentences like 'Karen believes Daniel is the killer'. We already defined both of those statements as part of the context in section 4.3. We will consider these statements to be simple facts that cannot be derived unless they are already part of the context\(^3\). We can however derive other statements from them, if we take both of the examples above to be true then we can deduce that Karen is not lying when she claims Daniel is the killer. Thus the sentence 'Karen is truthful about Daniel being the killer' can be derived. If we further know that Daniel is indeed the killer then we also know that the sentence 'Karen is right about Daniel being the killer' is true. The terms 'Is Truthful About', 'Is Lying About', 'Is Right About' and 'Is Wrong About' are defined in this way.

Definition 16.
The following axioms extend definition 9 with a definitions for belief and claim statements.

**Is Right About Derivation**

\[
\frac{c \models \Sigma p \quad c \models \Sigma 'Believe' p}{c \models \Sigma 'Is Right About' p} \quad \Sigma \models 'o.e.p.t'
\]

\[
\frac{c \not\models \Sigma p} {c \not\models \Sigma 'Is Right About' p} \quad \Sigma \models 'o.e.p.t'
\]

\[
\frac{c \not\models \Sigma 'Believe' p} {c \not\models \Sigma 'Is Right About' p} \quad \Sigma \models 'o.e.p.t'
\]

**Is Wrong About Derivation**

\[
\frac{c \not\models \Sigma p \quad c \models \Sigma 'Believe' p}{c \models \Sigma 'Is Wrong About' p} \quad \Sigma \models 'o.e.p.t'
\]

\[
\frac{c \models \Sigma p} {c \not\models \Sigma 'Is Wrong About' p} \quad \Sigma \models 'o.e.p.t'
\]

\[
\frac{c \not\models \Sigma 'Believe' p} {c \not\models \Sigma 'Is Wrong About' p} \quad \Sigma \models 'o.e.p.t'
\]

\(^3\) A number of rules were considered (and implemented) at first, but each either led to highly unintuitive result or was only relevant in very obscure cases and deemed not worth the effort so in the end this simple system was chosen.
A Coq module for natural logic

Is Truthful About Derivation

\[
\begin{align*}
  c \models \Sigma o \text{‘Say’ } p &\quad c \models \Sigma o \text{‘Believe’ } p \\
  \quad \Rightarrow \quad c \models \Sigma o \text{‘Is Truthful About’ } p
\end{align*}
\]

\[
\begin{align*}
  c \not\models \Sigma o \text{‘Say’ } p \\
  \quad \Rightarrow \quad c \not\models \Sigma o \text{‘Is Truthful About’ } p
\end{align*}
\]

\[
\begin{align*}
  c \not\models \Sigma o \text{‘Believe’ } p \\
  \quad \Rightarrow \quad c \not\models \Sigma o \text{‘Is Truthful About’ } p
\end{align*}
\]

Is Lying About Derivation

\[
\begin{align*}
  c \models \Sigma o \text{‘Say’ } p &\quad c \not\models \Sigma o \text{‘Believe’ } p \\
  \quad \Rightarrow \quad c \models \Sigma o \text{‘Is Lying About’ } p
\end{align*}
\]

\[
\begin{align*}
  c \not\models \Sigma o \text{‘Say’ } p \\
  \quad \Rightarrow \quad c \not\models \Sigma o \text{‘Is Lying About’ } p
\end{align*}
\]

\[
\begin{align*}
  c \models \Sigma o \text{‘Believe’ } p \\
  \quad \Rightarrow \quad c \not\models \Sigma o \text{‘Is Lying About’ } p
\end{align*}
\]

Originally we also tried to implement more global variants of these terms. The term ‘Is Truthful’, indicating that a person is truthful about all their claims in the current context for example. This turned out not to be possible though, because of the so called liar paradox. The liars paradox occurs when you allow sentence to refer to their own truth conditions, the classic example being the sentence ‘This sentence is false’. Sentences like that cannot be assigned a truth value because the sentence being true causes it to be false and vice versa. In our model, sentences like James ‘Is Wrong About’ (James ‘Is Wrong’) caused the same paradox, which we found out when we tried to model the definitions for these phrases in Coq.
Chapter 4

Implementation

4.1 Basic Types

We will introduce three basic types that will form the basis of the rest of the implementation. The first two types, $e$ and $t$ are just the basic types as laid out above, the third type $w$ is the implementation of contexts. We will briefly go over each these types.

4.1.1 $e$

The type $e$ refers to the basic entities in the language, it is implemented as an inductive type with a list of nullary constructors that represents the names in the language. As such any term of type $e$ (that we will consider, it is of course possible to make needlessly complicated terms in Coq that are of type $e$) is a single constant.

While natural language contains terms of type $e$ that are not specific names (pronouns, for instance)\(^1\), we will assume that in the translation from natural language to LLF the referents for these terms will be found and substituted. Note that so long as a pronoun always refers to the same entity in a given context it would be trivial to add this pronoun to the language, simply add a constructor for this pronoun to $e$ and provide rules for adding the necessary equivalences to a context. This would add very little to the language, though, as you would still have to find the corresponding entities before being able to translate said pronoun and it would needlessly complicate the definition of a context.

The full list of names in our implementation is:

\(^1\)Because of the way we use quantifiers, descriptions of things (‘The rock I found on the beach’) are not terms of type $e$, they are instead of type $(e \rightarrow t) \rightarrow t$
A Coq module for natural logic

Arthur : e
Bianca : e
Clark : e
Diana : e
Edward : e
Fiona : e
Gerald : e
Hannah : e
John : e
Mary : e

4.1.2 t

The other basic type is $t$, it is implemented as an inductive type with a constructor for each constant in the language (that isn’t a name). The type of these constructors depend on the kind of constant it represents: properties are of type $e \rightarrow t$, binary verbs are of type $e \rightarrow e \rightarrow t$, quantifiers are of type $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$, etc. The full list of constants that have been implemented is:

Creature : $e \rightarrow t$
Person : $e \rightarrow t$
Human : $e \rightarrow t$
Man : $e \rightarrow t$
Woman : $e \rightarrow t$
Student : $e \rightarrow t$
Bachelor : $e \rightarrow t$
Bird : $e \rightarrow t$
Lark : $e \rightarrow t$
Crow : $e \rightarrow t$
Penguin : $e \rightarrow t$
Dog : $e \rightarrow t$
Hound : $e \rightarrow t$

Move : $e \rightarrow t$
Fly : $e \rightarrow t$
Walk : $e \rightarrow t$
Run : $e \rightarrow t$
Sleep : $e \rightarrow t$
Snore : $e \rightarrow t$
Dream : $e \rightarrow t$

Who : $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow e \rightarrow t$
Implies : $t \rightarrow t \rightarrow t$
And : $t \rightarrow t \rightarrow t$
Or : $t \rightarrow t \rightarrow t$
Not : $t \rightarrow t$
Believe : $e \rightarrow t \rightarrow t$
IsRightAbout : $e \rightarrow t \rightarrow t$
IsWrongAbout : $e \rightarrow t \rightarrow t$
Say : $e \rightarrow t \rightarrow t$
IsTruthfulAbout : $e \rightarrow t \rightarrow t$
IsLyingAbout : $e \rightarrow t \rightarrow t$

4.1.3 w

The type $w$ models the concept of a context, it is implemented as a list of terms of type $t$. Elements in $w$ are ordered and the order is relevant when we evaluate consistency (see section 4.3), but in most cases it is still sufficient to think of contexts as a set.
4.2 Lexicographic Knowledge

We have two kinds of lexicographic knowledge that have to be modelled: lexicographic restrictions and lexicographical ordering. In this section we will go over the implementation of the knowledge itself, in the next we will explain how the validity of the appropriate axioms is guaranteed.

4.2.1 Lexicographic Ordering

The lexicographic ordering is a tree that determines which properties have to occur together. It is modelled as an inductive type $\text{Lex}$ that takes two terms of type $t$ and returns a $\text{Prop}$ (Coq’s internal representation of a proposition, basically a boolean) where $\text{Lex} (x \ o) (y \ o)$ models $x \subseteq y$. It has two constructors:

\[
\forall p \ q, \text{Simple} p \Rightarrow \text{Simple} q \Rightarrow \text{LexEdge} p \ q \Rightarrow \text{Lex} p \ q
\]

and

\[
\forall p \ q \ r, \text{Simple} p \Rightarrow \text{Simple} q \Rightarrow \text{Simple} r \Rightarrow \text{LexEdge} p \ q \Rightarrow \text{Lex} q \ r \Rightarrow \text{Lex} p \ r
\]

Here $\text{LexEdge}$ models the branches of the tree: given an $o$ of type $e$, $\text{LexEdge} (x \ o) (y \ o)$ is true if $x$ is a child node of $y$ in the lexicographic ordering. $\text{Simple}$ verifies that an LLF is simple.

We will always attempt to apply inductive types like $\text{Lex}$ and $\text{LexEdge}$ to terms of type $t$ even when, for instance, $e \rightarrow t$ would seem more appropriate at first glance. The reason for this is that if we have one hypothesis that tells us something about a term $x$ of type $e \rightarrow t$ and another hypothesis that states that $(x \ o) = (y \ o)$ then the first hypothesis doesn’t tell us anything about $y$. If on the other hand the first hypothesis told us something about $(x \ o)$ then we would know that the same thing applies to $(y \ o)$. Because such situations are fairly common we will try to talk about complete terms of type $t$ wherever possible.

4.2.2 Lexicographic Restrictions

Lexicographic restrictions are pairs of properties that cannot both apply to one object. This is implemented as an inductive type $\text{RestrictedCombination}$ which takes two terms of type $t$ and returns a $\text{Prop}$. For every lexicographic restriction between two properties $x$ and $y$ there is a constructor that looks like this:

\[
\forall o, \text{RestrictedCombination} (x \ o) (y \ o)
\]

Beyond that, there are two generic constructors that model the interaction between lexico-
graphic restrictions and the lexicographic ordering.

\[
\text{forall } p \ q \ r, \ \text{RestrictedCombination} \ p \ q \implies \text{Lex} \ r \ q \implies p \neq r \implies \text{RestrictedCombination} \ p \ r \\
\text{forall } p \ q \ r, \ \text{RestrictedCombination} \ q \ p \implies \text{Lex} \ r \ q \implies p \neq r \implies \text{RestrictedCombination} \ r \ p
\]

The absence of an explicit requirement that guarantees symmetry seems odd initially, however if we were to add something like:

\[
\text{forall } p \ q, \ \text{RestrictedCombination} \ q \ p \implies \text{RestrictedCombination} \ p \ q
\]

Then a hypothesis \text{RestrictedCombination} \ p \ q no longer provides useful information. For every such hypothesis it is possible that it was constructed using the above axiom, thus using inversion it yields (among other possibilities) \text{RestrictedCombination} \ q \ p which in turn yield \text{RestrictedCombination} \ p \ q as one of its possible constructions.

We still want lexicographical restrictions to be symmetric though, so we have a second inductive type \text{RestrictedCombination} that has the following two constructors:

\[
\text{RestrictedCombination} \ p \ q \implies p \neq q \implies \text{InvalidCombination} \ p \ q \\
\text{RestrictedCombination} \ q \ p \implies p \neq q \implies \text{InvalidCombination} \ p \ q
\]

Now, when we have a hypothesis of \text{InvalidCombination} \ p \ q we know that it was constructed using either \text{RestrictedCombination} \ p \ q or \text{RestrictedCombination} \ q \ p, but once we have taken this reduction step once we cannot take it again.

In each of the above constructors we have added a requirement that the two arguments for \text{InvalidCombination} and \text{RestrictedCombination} are different. This is technically redundant, in both cases this is already excluded by the rest of the constructor, but it simplified the proofs for some of the lemmas in section A.

4.3 Consistency

Now that we have a way to add lexicographic knowledge to the system we still have to guarantee that it is upheld. We do this in the by explicitly implementing the concept of a consistent context. Informally a context is consistent if it adheres to the axioms for lexicographic knowledge as well as quantifier monotonicity over ill defined quantifiers. The formal definition is slightly more elaborate.

4.3.1 Accessing a Context

As we said before, a context itself is nothing more than a list of LLFs and may contain any terms of type \( t \) that you can construct. However, we will only access a context through two functions \text{LiteralContains} and \text{Contains}. \text{LiteralContains} takes a context and an
LLF and returns true if that LLF is both contained in that context and represents a simple statement or has an ill defined quantifier as its outermost operator.

Contains is a wraparound function that is used to integrate who statements and the LiteralContains function. It takes a context and an LLF, if the LLF is a who statement it splits it into its conjuncts and recurses over them, otherwise it simply calls LiteralContains on it.

4.3.2 Consistent Contexts

Now we finally get to checking the consistency of a context. We do this through the Consistent function, this function takes a context $c$, if it is empty it returns True, if it is $p :: c$ it returns:

$$\neg\text{InvalidAddition } c_2 \land \text{ValidQuantifier } c_2 \land \text{Consistent } c_2$$

ValidQuantifier is a function used to specify additional requirements that ill defined quantifiers have to comply with, this is necessary to guarantee quantifier monotonicity between properly defined quantifiers and ill defined quantifiers.

InvalidAddition is where the heavy lifting happens, it checks for an LLF $p$ and a context $c$ whether $p :: c$ complies with the available lexicographic knowledge, given that $c$ does. It is defined as an inductive type with three constructors. The first constructor guarantees that nothing in $c$ has a lexicographical restriction with $p$:

forall $p \ q \ c$, \ LiteralContains c q \implies \neg \text{InvalidCombination } p q \implies \neg p \neq q \implies \text{InvalidAddition } c p$

The other two constructors ensure that $p$ doesn’t imply anything (that is part of a context) that isn’t already in $c$:

forall $p \ q \ c$, \ SimpleMonotonic p q \implies \neg \text{LiteralContains } c q \implies \neg p \neq q \implies \text{InvalidAddition } c p$
forall $p \ q \ c$, \ ComplexMonotonic p q \implies \neg \text{LiteralContains } c q \implies \neg p \neq q \implies \text{InvalidAddition } c p$

Here is where the ordering of the context matters, everything that is implied by $p$ has to occur in the context before $p$, otherwise we would not be able to check that the context was complete. This also requires that the lexicographical ordering is acyclic, because otherwise situations where $p$ has to occur before $q$ while $q$ has to occur before $p$ might happen.

This proved to be somewhat of an architectural hurdle during the implementation of who statements. Every who statement is derivable from both of its conjuncts and those conjuncts are derivable from it, this cyclical dependency made it impossible to add who sentences to the context, but (for reasons that we will get to in section 4.4) they could not be part of the derived constants either. The current implementation neatly sidesteps this problem by checking for who statements in the Contains function.
4.3.3 Simple Monotonicity

Simple monotonicity is checked by an inductive type `SimpleMonotonic`. It has a single fairly straightforward constructor:

\[
\forall p, q. \text{Lex} p q \Rightarrow \text{Simple} p \Rightarrow \text{Simple} q \Rightarrow p \neq q \Rightarrow \text{SimpleMonotonic} p q
\]

4.3.4 Complex Monotonicity

`ComplexMonotonic` implements the monotonicity rules for quantifiers. It has four fairly complex constructors:

\[
\forall z. (e \to t) \to (e \to t) \to t, \forall y_1 y_2 x o. \text{MonotonicQuantifier} (z y_1 x) \Rightarrow \text{MonotonicType} (z y_2 x) \uparrow \Rightarrow \text{MonotonicType} (z y_1 x) \uparrow \Rightarrow \text{Lex} (y_1 o) (y_2 o) \Rightarrow (x y_1 z) \neq (x y_2 z) \Rightarrow \text{ComplexMonotonic} (z y_1 x) (z y_2 x)
\]

And equivalent ones the other directions. There seems to be a lot going on here, so let’s break it down. What happens is that for two different LLFs \( z y_1 x \) and \( z y_2 x \) that have an ill defined quantifier \( z \) as there outermost operator, if \( y_1 \subseteq y_2 \) and \( z \uparrow \) then \( z y_1 x \) implies \( z y_2 x \). Thus, \( z y_1 x \) can’t be part of any consistent context unless \( z y_2 x \) is also part of that context and \( \text{ComplexMonotonic} (z y_1 x) (z y_2 x) \) holds.

4.4 Truth

The remainder of the semantics of the language is embedded in the inductive types `IsTrue` and `IsFalse`. After an initial phase where this functionality was implemented through a number of axioms\(^2\) `|=` and `=/=` are now implemented as two mutual inductive types that take a context and a statement and become `True` if and only if that statement is true/false in that context.

4.4.1 Mutual inductive types

A normal inductive type can use other inductive types in their constructors, but only if those other types are defined before it. As we want to define `IsTrue` and `IsFalse` in terms of each other we instead use mutual inductive types. Mutual inductive types consist of an inductive type that defines a second inductive type inside it’s definition, this allows us to simultaneously implement `IsTrue` and `IsFalse`, solving the issue of circular dependency.

\(^2\)This was dropped after it became clear that it was just to easy to model an inconsistent system and very difficult to find these inconsistencies.
As a consequence, the resulting inductive type has quite a substantial number of constructors. Most of them are direct implementations of the definitions given in section 3.5, though, so we will only go over them briefly.

### 4.4.2 Base

The first constructor of both `IsTrue` and `IsFalse` are implementations of basic truth/false derivation axioms:

\[
\forall c, p. \text{Consistent } c \implies \text{Contains } c, p \implies \text{IsTrue } c, p
\]

And its negative counterpart:

\[
\forall c, p. \text{Consistent } c \implies (\text{Simple } p \land \text{MonotonicQuantifier } p \land \text{IsWho } p) \implies \neg \text{Contains } c, p \implies \text{IsFalse } c, p
\]

Note that these constructors are not direct negations of each other, the requirement that a setting is consistent appears in both. As we will see below, this requirement appears in every constructor and reflects the idea that we only care about consistent contexts. In an inconsistent context every LLF is neither true nor false.

Also note that there is no direct implementation of `Cont(p)`. Instead we use \((\text{Simple } p \land \text{MonotonicQuantifier } p \land \text{IsWho } p)\), the reason for this is that similar requirements are quite common, but sometimes we want only `Simple` and `MonotonicQuantifier`, sometimes only `Simple` and `IsWho` and sometimes, as in this example, all three. Instead of defining three similar definitions we decided that it would be less confusing to just spell out which requirements we want at each point.

### 4.4.3 Boolean Operators

There is not much to say about the implementation of boolean operators, they are all more or less direct implementations of their respective axioms. We did try to minimize the number of necessary constructors by combining axioms that implement a disjunction. Thus we have

\[
\forall c, p, q. \text{Consistent } c \implies \text{IsTrue } c, p \lor \text{IsTrue } c, q \implies \text{IsTrue } c, (p \lor q)
\]

which implements both of the positive axioms for `Or`. The same thing was done for `Implies` and the negation of `And`.
4.4.4 Quantifiers

The quantifiers differ from their axioms a bit more, so we will go over each of them separately.

\(A\)

The implementation of \(A\) differs from its axioms in two ways, here is the positive constructor:

\[
\forall c \text{ o } x \text{ y, Consistent } c \implies (\text{Simple } (x \text{ o}) \land \text{IsWho } (x \text{ o})) \implies \\
(\text{Simple } (y \text{ o}) \land \text{IsWho } (y \text{ o})) \implies \text{Contains } c (x \text{ o}) \implies \text{Contains } c (y \text{ o}) \implies \text{IsTrue } c (A x y)
\]

The axiom only specified that \(x\) and \(y\) should be of type \(e \to t\), but in the implementation we require that they are either simple or who statements. This makes certain proofs considerably less difficult at the cost of losing the ability to use custom terms of type \(e \to t\) as arguments for \(A\).

The other deviation from the axioms is that we directly check whether \(x \text{ o}\) and \(y \text{ o}\) are part of the context. This is simply more expedient as the only way a simple statement or a who statement can be true is if it is part of the context.

\(\text{Some}\)

In an effort to reduce the number of constructors, \(\text{Some}\) is implemented as a definition, stating that it is equivalent to \(A\).

\[
\text{Definition Some } (x: e \to t) \ (y: e \to t): \ t := A x y.
\]

\(\text{Each}\)

Except for the deviations already mentioned for \(A\), \(\text{Each}\) is a direct implementation of the definition.

\(\text{The}\)

Aside from the already mentioned deviations, the negative axioms for \(\text{The}\) were merged into one constructor and we used logical ands and ors rather than the LLF versions. Again, both to make the definition more direct and reduce the number of constructors.
A Coq module for natural logic

No

No is a direct implementation of the specification.

4.4.5 Beliefs

Belief statements are mostly direct implementations of the corresponding axioms. The only deviations come from the already mentioned combining of axioms into disjunctions and, again, checking whether the basic belief statements are part of the context rather than whether they are true.

4.5 LTac Commands

Alongside the model, several Ltac tactics were implemented to facilitate writing proofs concerning LLFs; a full list is in appendix B but we will go over the most prominent ones here.

4.5.1 Knowledge

The way the model is set up means that many proofs end up branching out into many trivial cases that nevertheless each require some steps to solve. This is where the Knowledge tactic comes in. It combines a number of more specialized tactics in an attempt to immediately solve the current goal.

Originally it was set up to solve every case that could be solved using only lexicographic knowledge, but over time it expanded into a catch all tactic for the various trivial cases that often come up during proofs about LLFs. Currently it automatically solves the following cases:

- Any case that can be solved exclusively with knowledge of the lexicographic ordering
- Most cases that can be solved exclusively with knowledge of lexicographic restrictions
- Any case that has a false hypothesis that claims some LLF is a simple statement, a who statement or an ill defined quantifier statement.
- Any case that can be solved by applications of the native simpl, discriminate and intuition tactics.
- Various other specialized cases that are generated by calling Convert on specific hypothesis.
Knowledge never damages the proof state if it fails, it either solves the goal instantly or leaves the proof state intact.

### 4.5.2 Monotonicity

The Monotonicity tactic is used to immediately solve any case that can be solved with monotonicity. That is, if the goal can be derived from one or more of the hypothesis using only monotonicity steps, Monotonicity will instantly complete the goal.

Internally, if the goal is a who statement it first splits it into its conjuncts. Then if the goal is of the form \(\text{Contains } c p\) it will attempt to solve it using by applying the lemma L\_SimpleMonotonicity followed by repeated applications of Knowledge. L\_SimpleMonotonicity is one of the lemmas that were proven in order to verify that the system adheres to the specification (see section 5.2), it states:

\[
\text{forall } c p q, \text{Consistent } c \implies \text{SimpleMonotonic } p q \implies \text{Contains } c p \implies \text{Contains } c q.
\]

If the goal is of a different shape, Monotonicity calls another tactic based on the outermost operator (if it is a quantifier) in the goal. These other tactics all are very similar, so we will only show the one for \(A\). The tactic used for this MonotonicityA:

```
Ltac MonotonicityA:=
  match goal with
  | [ \[H: \text{IsTrue } ?(A ?P ?Q) \implies \text{IsTrue } ?(A ?P)\]] \implies \text{apply L\_MonotonicityA1 with } (x1 := P); Knowledge
  | [ \[H: \text{IsTrue } ?(A ?P ?Q) \implies \text{IsTrue } ?(A ?P)\]] \implies \text{apply L\_MonotonicityA2 with } (y1 := Q); Knowledge
  | [ \[H: \text{IsTrue } ?c (A ?P) \implies \text{IsTrue } ?c (A ?P)\]] \implies \text{assert (IsTrue } c (A P Q));\text{try MonotonicityA}
  end.
```

It has three cases, first it checks whether the current goal can be solved directly by monotonicity on the first or second argument. If this is the case then it solves the goal by applying the correct Lemma. Otherwise it looks for a suitable next step, makes the appropriate assertion and recurses.

### 4.5.3 Convert

The basic Convert tactic, when called on a hypothesis IsTrue \(c p\), converts that hypothesis into a more basic form, thus IsTrue \(c ((A\text{ Man Walk}) \text{ And } (A\text{ Woman Walk}))\) becomes IsTrue \(c (A\text{ Man Walk}) \land \text{IsTrue } c (A\text{ Woman Walk})\) and IsTrue \(c (A\text{ Man Walks})\) splits into Simple \(\text{Man o}\) \(\lor\) IsWho \(\text{Man o}\), Simple \(\text{Walk o}\) \(\lor\) IsWho \(\text{Walk o}\), Contains \(c (\text{Woman o})\) and Contains \(c (\text{Walk o})\). Internally this tactic calls inversion on the hypothesis in question and resolves some of the more trivial cases this generates, leaving the
user to solve the interesting cases.

There are two variants of this tactic: ConvertGoal, which does the same as Convert\(^3\) but works on the goal instead of a hypothesis and ConvertAll, which is a powerful catch all tactic that intelligently combines Convert, ConvertGoal, Knowledge and a number of other specialized tactics to reduce an LLF to its most basic state, in many cases completing the proof outright. ConvertAll and ConvertGoal are not entirely foolproof, there are cases where they make the wrong choices in which hypothesis to apply, throwing away information that is required to complete the proof. In these cases falling back to less advanced tactics or even directly applying the constructors of IsTrue might be necessary. These case are fairly rare, though and almost all the proofs in both test scenarios can be completed without knowledge of the structure of IsTrue.

4.5.4 SolveConsistency

Manually verifying the consistency of a context is a very tedious process that involves making the same few steps over and over. The tactic SolveConsistency was implemented to automatically go through this process. It first fully unfolds the context and then verifies for each element that it doesn’t conflict with any of the other elements and that it doesn’t have any implications that are not themselves part of the context.

This is a very time consuming process, applying the tactic to a context containing only six elements takes several minutes to complete. During the verification process we usually created separate lemmas for every context that needed to be verified and then used SolveConsistency on those lemmas once, commenting it out and replacing it with Admitted afterwards.

---

\(^3\)From the users perspective at least, internally it looks up the correct axiom to apply to the goal, then applies it and attempts to solve some of the more trivial cases that this generates.
Chapter 5

Verification

5.1 Introduction

In this chapter we will go over some of the methods we have used to ensure that the model we constructed in the previous chapter is correct. We have used three methods for this purpose: first we established a number of lemmas that we knew the model should adhere to (things like excluded middle). Secondly, the prolog theorem prover mentioned in section 1.1 contains a test suite used during development to verify that the implementation is correct. We converted this test suite to Coq and verified that these tests could be completed. Finally, we designed an additional number of tests to provide a good cross section of the functionality of the model and verified that these tests could be completed without problems.

5.2 Lemmas

The first verification step was to establish a number of lemmas that we knew the model should adhere to. The full list of the lemmas that we proved is in appendix A, but we will go over the two most prominent ones here.

Many of the Lemmas that we have proved seem trivial at first glance, but because we defined true and false separately it is not guaranteed that they hold (and, in fact, many times during the project small changes to the model falsified some of these lemmas).

5.2.1 TrueNotFalse and FalseNotTrue

A mayor focus for this part of the verification process was to guarantee that the interaction between IsTrue and IsFalse conforms to the concepts of True and False. The first part of
A Coq module for natural logic

this is to verify that no statement is both true and false, to do this, we first proved that if a statement is true then it cannot be false, after that, the inverse is trivial.

Formally the lemma that we want to prove is:

\[ \text{Lemma L\_TrueNotFalse: } \forall c \, p, \text{Consistent } c \Rightarrow \text{IsTrue } c \, p \Rightarrow \neg \text{IsFalse } c \, p. \]

The proof goes by induction over \( p \), for every constructor we got a proof state that looked something like this:

- \( H: \text{Consistent } c \)
- \( H_0: \text{IsTrue } c \, (p_1 \, \text{And} \, p_2) \)
- \( H_1: \text{IsFalse } c \, (p_1 \, \text{And} \, p_2) \)
- \( \text{IHp1: } \text{IsTrue } c \, p_1 \Rightarrow \text{IsFalse } c \, p_1 \Rightarrow \text{False} \)
- \( \text{IHp2: } \text{IsTrue } c \, p_2 \Rightarrow \text{IsFalse } c \, p_2 \Rightarrow \text{False} \)

We mostly ignored the induction hypothesis and focused on proving a contradiction between \( H_0 \) and \( H_1 \). From there, the proof was largely a proof by construction, we specifically designed the constructors in such a way that the conditions for the \texttt{IsFalse} constructor form a contradiction with those for the \texttt{IsTrue} constructor.

5.2.2 Excluded Middle

The other lemma that we will discuss here is excluded middle, formally:

\[ \text{Lemma L\_ExcludedMiddle: } \forall c \, p, \text{Consistent } c \Rightarrow \text{IsTrue } c \, p \lor \text{IsFalse } c \, p. \]

Again, we start by doing induction over \( p \), after getting rid of some trivial base cases we are left with one goal for each constructor of \texttt{IsTrue}.

For the boolean operators, the proof from there is just case distinction on the values of their parameters. For example, the proofstate for \texttt{And} looks like this:

- \( H: \text{Consistent } c \)
- \( H_0: \text{IsTrue } c \, (p_1 \, \text{And} \, p_2) \)
- \( H_1: \text{IsFalse } c \, (p_1 \, \text{And} \, p_2) \)
- \( \text{IHp1: } \text{IsTrue } c \, p_1 \lor \text{IsFalse } c \, p_1 \)
- \( \text{IHp2: } \text{IsTrue } c \, p_2 \lor \text{IsFalse } c \, p_2 \)
- \( \text{IsTrue } c \, (p_1 \, \text{And} \, p_2) \lor \text{IsFalse } c \, (p_1 \, \text{And} \, p_2) \)

We can \texttt{destruct} \text{IHp1} and \text{IHp2}, giving us four cases corresponding to the four possible values of \( p_1 \) and \( p_2 \). We can then easily show that the goal holds for each of those cases.
For the rest of the constructors the induction hypotheses don’t tell us much so we use Coq’s classical logic module, Classical, that gives us the lemma classic. This lemma allows us to prove for every \( p \) that \( p \land \neg p \) holds (note that this is Coq’s internal negation). Take \( A \) as an example again, we want to prove \( \text{IsTrue} c (A \ t0 \ t1) \lor \text{IsFalse} c (A \ t0 \ t1) \). By selecting appropriate values for \( p \) we can then transform the proofstate such that we have to prove \( \text{IsFalse} c (A \ t0 \ t1) \) from the negation of \( \text{IsTrue} c (A \ t0 \ t1) \) and the constructors are designed in such a way that this is always possible (if not always very simple).

5.3 Prolog Test Suite

The first series of tests we used to verify the correctness of the model came from the prolog theorem prover by Lasha Abzianidze. His implementation contained a test suit that (among other things) contained 35 inferences that should be proved to be either true or not true. The full list is in appendix C.1.

The way the prolog prover works is that it takes two LLFs and either tells you whether one can be derived from the other in all contexts\(^1\). As a consequence, all the tests in the test suite when converted to Coq consist of an LLF composed of the two LLFs from the test suite combined by an implies statement. We then either have to prove that this LLF is a tautology (if the inference is valid) or prove that there is a context in which it doesn’t hold (if the inference is invalid).

Most tests can be completed by a single call to ConvertAll plus one or two other tactics. We will look at one of those that took a little more effort.

\[
\text{IsTrue} \ TW ((\text{Sleep John And Sleep Mary}) \implies ((\text{Some Man Sleep}) \ And \ (\text{Some Woman Sleep}))).
\]

In the prolog prover, facts like John is a man and Mary is a woman are hardcoded in the model, thus the above sentence is a tautology. In our system those facts would be part of the context and so sentences like this technically no longer are a tautology, this on, for example, is only true if John and Mary are humans of opposite gender. To fix this we created a context \( TW \) (test world) that contains those facts and proved these LLFs for that context.

The proof itself starts by applying a previously proven Lemma that proofs the consistency of \( TW \), we could have just used SolveConsistency here, but that tactic is really slow so we don’t want to use it in every proof where we use TW. After this we replace every instance of \( \text{Some} \) by \( A \). Then we repeatedly apply ConvertGoal to break the goal down, which leaves us with the following proof state:

---
\(^1\)Actually, the prolog theorem prover works differently from our implementation, it contains no concept of contexts, instead it has a small amount of hardcoded background knowledge and applies transformation rules to attempt to reduce the source LLF to the target LLF. If it succeeds it returns true, otherwise it returns false.
H: Consistent TW
H0: Sleep John And Sleep Mary
1) IsTrue TW (A Man Sleep)
2) IsTrue TW (A Woman Sleep)

The first goal can be solved using ConvertAll. For the second one we first have to convert
H0: Sleep John And Sleep Mary into IsTrue TW (Sleep John) and IsTrue TW (Sleep
Mary). This is the point where ConvertAll makes a mistake, it tries to resolve IsTrue TW (A
Woman Sleep) by finding a woman that sleeps, and selects John to be that woman (it can’t
look into TW to observe that this isn’t the case). After we dropped IsTrue TW (Sleep John)
from the list of hypothesis, using the clear tactic, the rest of the proof can be completed
using ConvertAll.

5.4 Testscenarios

As a final verification step we wrote a number of test scenarios of our own, again the full list
is in the appendices (appendix C). Most of these test can be solved by a singe application
of ConvertAll plus a few bookkeeping tactics. This is one of the tests that requires a little
more effort:

Tautology ((Each (Creature Who Move) Crow) Implies (No (Person Who Fly) Student)).

(Tautology p is a shorthand for forall c, Consistent c \implies IsTrue c p.) The problem
with this test is that it uses the fact that no entity can be both a Bird and a Human while
not using either of those terms. This isn’t really a problem for the model, it is fairly easy
to prove, but it falls outside of the scope of both the convert tactics and Monotonicity so it
takes some more in-depth knowledge of the model.

For the actual proof we first use ConvertAll to decompose the lemma, this gives us the
following proof state:

H2: \forall o : e, Contains c ((Creature Who Move) o) \implies Contains c (Crow o)
H0: Consistent c
H: Contains c (Person o)
H1: Contains c (Fly o)
H6: Contains c (Student o)
False

We want to use H2 to prove that o is a crow. We can do this by first asserting Contains
c (Crow o) and then applying H2, this gives us the subgoal Contains c ((Creature Who
Move) o) which can be completed using Monotonicity. The restriction we want to use is
between Person and Bird so we also have to prove Contains c (Bird o), we do this by
asserting it and using Monotonicity again.
Now we want to prove a contradiction exists between \texttt{Contains \(c \ (\text{Person} \ o)\)} and \texttt{Contains \(c \ (\text{Bird} \ o)\)}. In order to do this we first assert that \((\text{Person} \ o)\) could not have been an invalid addition to \(c\) (\(\sim \text{InvalidAddition} \ c \ (\text{Person} \ o)\)). This is true because \(c\) contains \((\text{Person} \ o)\) and is consistent. There is a lemma, \texttt{L\_NoInvalidElement}, that states exactly this so that part is easy. Next we want to prove that \((\text{Person} \ o)\) is an invalid addition to \(c\), we do this by applying the first constructor of \texttt{InvalidAddition} \((\text{LiteralContains} \ c \ q \ \Rightarrow \text{InvalidCombination} \ p \ q \ \Rightarrow p \neq q \Rightarrow \text{InvalidAddition} \ c \ p)\) with \((\text{Bird} \ o)\). \texttt{Knowledge} sorts out the rest of this subgoal. Now we have two hypotheses that are a direct contradiction of one another so the only thing that is left is use the \texttt{contradiction} tactic to finish the proof.
Chapter 6

Future Work

There are a few extensions of the model that we would have liked to implement, but are not part of the current model for one reason or another.

6.1 Quantifier Monotonicity

Monotonicity on arguments is working in the current model, but there are two cases where monotonicity between quantifiers does not fully function. Currently quantifier monotonicity between two well defined quantifiers (The bird flies therefore a bird flies) and from an ill defined quantifier to a well defined quantifier (Many birds fly therefore a bird flies) work fine. But it has proved to be very difficult to implement monotonicity relations from a well defined quantifier to an ill defined quantifier (Every bird flies therefore most birds fly) and between ill defined quantifiers (theoretically possible but no examples using the currently implemented quantifiers).

Monotonicity between ill defined quantifiers is possible within the current system and was actually partly implemented but dropped due to time constraint and because there are no examples within the current model. Monotonicity from a well defined quantifier to an ill defined quantifier requires significant revisions of the way contexts currently work (at the level where context consistency is computed there is no notion of well defined quantifiers). There are ways around this though and figuring out a way to make this work would be an interesting challenge. Given more time we would have liked to attempt it.

6.2 Improving the tactics

The currently implemented tactics perform very well in the vast majority of cases, but there are several situations where the user still has to put significant effort into guiding the process.
Ideally we would have liked `ConvertAll` followed by `Monotonicity` to instantly complete every possible proof and while this combination does solve the vast majority of cases, there are still cases such as the one from section 5.4 that require additional user input. We would have liked to fix this.

### 6.3 Extending the vocabulary

The obvious extension for the system is adding more functionality by increasing the size of the subset of language that it can model. A few good possibilities are non declarative sentences such as questions or imperatives, temporal logic (before, after etc), pronouns and other kinds of lexicographical knowledge as well as many smaller additions like more quantifiers and other kinds of verbs and nouns.
Appendix A

Full list of verified Lemmas

\textbf{NotContainsEmpty:} \forall p, \neg \text{Contains Empty } p.

\textbf{ContainsSimpleElements:} \forall c p, \text{Consistent } c \implies \text{Contains } c p \implies \text{Simple } p \lor \text{MonotonicQuantifier } p \lor \text{IsWho } p.

\textbf{TrueNotFalse:} \forall c p, \text{Consistent } c \implies \text{IsTrue } c p \implies \neg \text{IsFalse } c p.

\textbf{FalseNotTrue:} \forall c p, \text{Consistent } c \implies \text{IsFalse } c p \implies \neg \text{IsTrue } c p.

\textbf{ExcludedMiddleForA:} \forall c x y, \text{Consistent } c \implies \text{IsTrue } c (A x y) \lor \text{IsFalse } c (A x y).

\textbf{ExcludedMiddle:} \forall c p, \text{Consistent } c \implies \text{IsTrue } c p \lor \text{IsFalse } c p.

\textbf{ConvertNot:} \forall c p, \text{Consistent } c \implies \text{IsTrue } c (\text{Not } p) \implies \neg \text{IsTrue } c p.

\textbf{RevertNot:} \forall c p, \text{Consistent } c \implies \neg \text{IsTrue } c p \implies \text{IsTrue } c (\text{Not } p).

\textbf{NotContainsStep:} \forall c p q, \neg \text{Contains } (c \text{ +++ } p) q \implies \neg \text{Contains } c q.

\textbf{NIEHelp:} \forall c q p, \text{InvalidAddition } (c \text{ +++ } q) p \implies \neg (\text{InvalidCombination } p q) \implies \text{InvalidAddition } c p.

\textbf{NoInvalidElements:} \forall c p, \text{Consistent } c \implies \text{LiteralContains } c p \implies \neg \text{InvalidAddition } c p.

\textbf{LitContainsNoWho:} \forall c p, \text{LiteralContains } c p \implies \neg \text{IsWho } p.

\textbf{SimpleMonotonicity:} \forall c p q, \text{Consistent } c \implies \text{SimpleMonotonic } p q \implies
A Coq module for natural logic

Contains \(c p \implies \text{Contains} \ c \ q.\):

**ComplexMonotonicity:** \(\forall \ c \ p \ q, \text{Consistent} \ c \implies \text{ComplexMonotonic} \ p \ q \\implies \text{Contains} \ c \ p \implies \text{Contains} \ c \ q.\):

**WhoMonotonicity1:** \(\forall \ c \ x \ y \ o, \text{Consistent} \ c \implies \text{Contains} \ c \ (x \ o) \\implies \text{Contains} \ c \ (y \ o) \implies \text{Contains} \ c \ ((x \ \text{Who} \ y) \ o).\):

**WhoMonotonicity2:** \(\forall \ c \ x \ y \ o, \text{Consistent} \ c \implies \text{Contains} \ c \ ((x \ \text{Who} \ y) \ o) \implies \text{Contains} \ c \ (x \ o) \land \text{Contains} \ c \ (y \ o).\):

**MonotonicityEach1:** \(\forall \ c \ x \ y \ o, \text{Consistent} \ c \implies \text{Contains} \ c \ (x \ o) \land \text{Contains} \ c \ (y \ o).\)

**MonotonicityEach2** \(\forall \ c \ x \ y \ o, \text{Consistent} \ c \implies \text{Contains} \ c \ (x \ o) \land \text{Contains} \ c \ (y \ o).\)

**ImpTrans:** \(\forall \ c \ x \ z \ y, \text{Consistent} \ c \implies \text{IsTrue} \ c \ (x \ \text{Implies} \ y) \implies \text{IsTrue} \ c \ (y \ \text{Implies} \ z) \implies \text{IsTrue} \ c \ (x \ \text{Implies} \ z).\)

**ImpRefl:** \(\forall \ c \ p, \text{Consistent} \ c \implies \text{IsTrue} \ c \ (p \ \text{Implies} \ p).\)

**ImpliesHyp:** \(\forall \ c \ p \ q, \text{Consistent} \ c \implies \text{IsTrue} \ c \ p \implies \text{IsTrue} \ c \ q \implies \text{IsTrue} \ c \ (p \ \text{Implies} \ q).\)
NoSelfRestrictions: \( \forall p, \neg \text{RestrictedCombination} \ p \ p. \)

MonotonicityAWho: \( \forall x \ y \ z \ c, \text{Consistent} \ c \ \Rightarrow \ ( \forall o, \text{Simple} \ (x \ o) \ \vee \ \text{IsWho} \ (x \ o)) \ \Rightarrow \ \text{IsTrue} \ c \ (\forall x \ \text{Who} \ y \ z) \ \Rightarrow \ \text{IsTrue} \ c \ (\forall x \ z). \)

SplitWho1: \( \forall x \ y \ o \ c, \text{Contains} \ c \ ((x \ \text{Who} \ y) \ o) \ \Rightarrow \ \text{Contains} \ c \ (x \ o) \ \land \ \text{Contains} \ c \ (y \ o). \)

SplitWho2: \( \forall x \ y \ o \ c, \text{Contains} \ c \ (x \ o) \ \Rightarrow \ \text{Contains} \ c \ (y \ o) \ \Rightarrow \ \text{Contains} \ c \ ((x \ \text{Who} \ y) \ o). \)
Appendix B

Full list of LTac Commands

**Convert**: Simplifies a hypothesis.

**ConvertGoal**: Simplifies the goal.

**UnfoldTautology**: Unfolds tautology and introduces the new variable.

**ConvertAll**: Simplifies the proofstate as much as possible, potentially completing the proof outright.

**SplitWho**: Splits a \texttt{Who} statement into its conjuncts.

**Knowledge**: Resolves a lot of simple cases Automatically

**LexAuto**: Automatically solves all cases for \texttt{Lex}.

**SimpleAuto**: Automatically verifies that an LLF is simple.

**ComplexAuto**: Automatically verifies that an LLF has an ill defined quantifier as outermost operator.

**LexStep**: Solves more complicated cases for \texttt{Lex}.

**CleanUp**: Removes redundant hypotheses.

**BasicLex**: Solves the base cases for \texttt{Lex}.

**SolveNo**: Solves annoying situation resulting from converting \texttt{No}.

**Monotonicity**: Solves problems that can be resolved using only monotonicity.
MonotonicityA: Solves monotonicity for A.

MonotonicityNo: Solves monotonicity for No.

MonotonicityEach: Solves monotonicity for Each.

SolveQuantifier: Solves cases relating to direction of monotonicity.

DiscrQuantifier: Solves cases with an equivalence between two unequal quantifier statements in the hypotheses.

ResolveComplexMonotonicity: Solves cases with quantifier monotonicity

MonotonicityRegular: Solves cases with base monotonicity.

SolveConsistency: Automatically proves that a context is consistent.

PopContains: Unwraps the next contains statement from the context.

NextLex: Begin the next step in the lexicographic ordering.

SolveLex: Solve the current step in the lexicographic ordering.

SolveRestrictedCombination: Solve the cases for consistent contexts that revolve around restrictions.

SolveInvalidCombination: Solve the base cases for consistent contexts that revolve around restrictions.

SolveContains: Solve the cases for consistent context that revolve around Contains statements.

SolveSimpleMonotonic: Solve the cases for consistent context that revolve around simple monotonicity.

SolveComplexMonotonic: Solve the cases for consistent context that revolve around complex monotonicity.

SolveCombine: Combines the above tactics.
Appendix C

Full list of Tests

C.1 Prolog Tests

Lemma Infer1: ∀ o, Tautology (Not((Man o) Or (Woman o)) Implies (Not(Man o) And Not(Woman o))).

Lemma Infer2: ∀ o, Tautology (Not((Man o) And (Woman o)) Implies (Not(Man o) Or Not(Woman o))).

Lemma Infer3: ∀ o, Tautology ((Not(Man o) And Not(Woman o)) Implies Not((Man o) Or (Woman o))).

Lemma Infer4: ∀ o, Tautology(Not (Man o) Or Not (Woman o)) Implies Not((Man o) And (Woman o))).

Lemma Infer5: ∀ o, Tautology ((Lark o) Implies (Bird o)).

Lemma Infer6: ∀ x, Tautology (( Not ((Man x) And (Woman x))) Implies (Not (Man x) And Not (Woman x))).

Lemma Infer7: ∀ o, Tautology (Not ((Man o) Or (Woman o)) Implies (Not (Man o) Or Not (Woman o))).

Lemma Infer8: ∀ x, Tautology ((Bird x) Implies (Lark x)).

Lemma Infer9: Tautology (A Bird Fly Implies A Lark Fly).
Lemma Infer10: Tautology (A Lark Fly Implies A Bird Move).


Lemma Infer12: Tautology (Each (Person Who Touch Diana) Run Implies (Most (Student Who Kiss Diana) Move)).

Lemma Infer13: Tautology (No Bird Move Implies (No Lark Fly)).

Lemma Infer14: ∀ o, Tautology (((o Kiss Fiona) And (Student o) Implies ((o Touch Fiona) And (Person o)))).

Lemma Infer15: ∀ x, Tautology (If (Woman x) Then (Man x) Implies If (Not (Woman x)) Then (Not (Man x))).

Lemma Infer16: Tautology (Sleep John And Not (Sleep John) Implies (Walk John)).

Lemma Infer17: Tautology (If Sleep John Then Snore John Implies (If Not (Snore John) Then Not (Sleep John))).

Lemma Infer18: Tautology (If Sleep John Then Snore John Implies ((Snore John) Or (Not (Sleep John)))).

Lemma Infer19: Tautology (Snore John Implies Sleep John).

Lemma Infer20: Tautology (Most Man Walk Implies (Most Man Move Or Each Woman Move)).

Lemma Infer21: Tautology (Some Bachelor Sleep Implies (Some Man Sleep Or Few Person Sleep)).

Lemma Infer22: ∀ x, Tautology (Most Woman x Implies A Woman x).

Lemma Infer23: Tautology (Some Bachelor Sleep Implies Some Man Sleep).

Lemma Infer24: Tautology (Some Dog Run Implies (Some Hound Move Or Most Creature Run)).

Lemma Infer25: Tautology (Some Hound Run Implies (Some Dog Run Or Most Creature Run)).

Lemma Infer26: Tautology (Some Dog Run Implies (Some Man Move Or Most Creature Sleep)).

Lemma Infer27: Tautology (((Some Dog Run And Some Bachelor Sleep) Implies
A Coq module for natural logic

(Some Man Sleep)).

Lemma Infer28: Tautology ((Some Bachelor Sleep And Some Dog Run) Implies (Some Man Sleep)).

Lemma Infer29: IsTrue TW (Sleep John Implies (Some Man Sleep)).

Lemma Infer30: IsTrue TW ((Sleep John And Sleep Mary) Implies ((Some Man Sleep) And (Some Woman Sleep))).

Lemma Infer31: IsTrue TW ((Sleep John) Implies (Some Person Sleep)).

Lemma Infer32: IsTrue TW ((No Person Sleep) Implies (Not (Sleep John))).

Lemma Infer33: IsTrue TW ((Each Person Walk And No Person Fly) Implies (Walk John And Not (Fly Diana))).

Lemma Infer34: IsTrue TW ((Each Man Walk) Implies (Walk John)).

Lemma Infer35: IsTrue TW ((No Person Fly And Each Man Walk) Implies (Walk John)).

C.2 Test Scenarios

Lemma test_1: Tautology((A Woman Walk) Implies (A Woman Walk)).

Lemma test_2: Tautology (A Crow Fly Implies (A Creature Move)).

Lemma test_3: Tautology (No Human Sleep Implies (No Student Dream)).

Lemma test_4: IsTrue TW ((Walk Clark) Implies (A Person Move)).

Lemma test_5: IsTrue TW (Each Person (Love Diana) Implies (Each Man (Like Diana))).

Lemma test_6: IsTrue TW (Fly Arthur Implies A Bird Move).

Lemma test_8: IsTrue TW (Fly Arthur Implies (Not (No Creature Move))).
Lemma test.9: Tautology (Each (Person Who Walk) Move).

Lemma test.10: Tautology ((Each (Creature Who Move) Bird) Implies (No (Person Who Fly) Student)).

Lemma test.11: Tautology ((Not (Most Man Sleep)) Implies (Not (Most Man Dream))).

Lemma test.12: ∀ c, Consistent c ⇒ (Edward Is Truthful In c / Edward Is Right In c) ⇒ IsTrue c ((Edward Say (Fly Hannah)) Implies (Fly Hannah)).

Lemma test.13: Tautology (Edward Say (Fly Hannah) And (Edward Is Lying About (Fly Hannah)) And (Edward Is Wrong About (Fly Hannah)) Implies (Fly Hannah)).

Lemma test.14: Tautology ((Edward Kill Fiona) Implies (Edward Hurt Fiona)).

Lemma test.15: IsTrue TW (Not(Walk Edward And No Person Move)).

Lemma test.16: Tautology (Few Bird Fly Implies Few Penguin Fly).

Lemma test.17: IsTrue TW (((Fiona Kill Hannah) Or (Edward Kill Hannah)) And (The (Person Who (Kill Hannah)) Man) Implies (Edward Kill Hannah)).

Lemma test.18: Tautology ((Edward Kill Hannah) Implies ((Person Edward) Implies ((Person Who ((Kill Hannah))) Edward))).

Lemma test.19: ¬Tautology ((A Person Walk)).

Lemma test.20: ¬Tautology ((No Person Walk)).
Bibliography


