A Coq module for natural logic

Rutger Huijben

September 3, 2013

Contents

Chapter 1

Introduction

Many formal logic systems originally have their roots in natural language, $\neg$ $\wedge$ $\forall$ 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[7], 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 sentences 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 prooftree 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, 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:

$$\neg\left((A \Rightarrow \neg B) \land (A \Rightarrow B) \land A\right).$$

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 consist 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:

```

______________________________
 ~((A → ~B) ∧ (A → B) ∧ 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 `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. $\sim A$ is actually `A → False`). Now the proofstate looks like this:

```
 H:((A → ~B) ∧ (A → B) ∧ A)
______________________________
 False
```

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

```
 H: A → ~B
 H0:  A → B
 H1:  A
_______________
 False
```

The next tactic we are going to use is `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 `apply H` (remember that negations are just implications with false as the right-hand side) we get the following proof state:

```
 H: A → ~B
 H0:  A → B
 H1:  A
_______________
 1)A
_______________
 2)B
```

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 `assumption`. Coq then automatically switches to the second goal. Here we will again use `apply`, but this time we will use it on `H1` instead of on the goal. This is done by calling `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 `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, `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 funtional 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.

```
Inductive tree :  Set :=
 | leaf :  tree
 | node :  tree -> nat -> tree -> tree.
```

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 `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).

Chapter 2

Philosophical Backgrounds

2.1 History

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[6] (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.

2.2 Related Work

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*[9] 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 quantifier 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 [8]. 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 LLF, 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 Retorés 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.

2.3 Philosophical relevance

One question that we felt was important to address here is the question of the philosophical relevance of this project. This question, upon examination, splits into two separate question, firstly, what is the philosophical relevance of LLFs? And secondly, what is the philosophical relevance of formalizing the specification of a formal language to such a degree as is done in this project?

The answer to the first question is probably self evident to most philosophers of language. Questions in philosophy very often come down to analysing the question rather than looking for an answer, it therefor stands to reason that anything that gives us a better understanding of the language we use will give us an edge in understanding philosophical questions. Formalizing natural language plays a big role in this and LLFs provide a valuable new insight in this area by directly using the syntax of lambda calculus in this process.

The answer to the second question is probably self evident to most computer scientists. The human mind has an exceptional capability of deluding itself into thinking that we have found the solution to a problem when the basic outlines of the solution seem to fit the basic outlines of the problem. After that, we usually only test this solution on those parts of the problem that it is designed to fit. To combat this blind spot in human reasoning it is useful to make a formal description of that solution that can be thoroughly tested. By implementing LLFs into Coq we are forced to provide a concrete and complete definition which gives a greater degree of insight than applying it in a more informal context would.

Chapter 3

Specification and Implementation

3.1 Introduction

In this chapter we give an overview of the system as well as an indication of how the various parts are implemented. A comprehensive specification of the system is beyond the scope of this paper (for this, see the version of this paper for the TU/e[5]), but we will attempt to give a good high level overview of what the system is capable of and how we achieved this.

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 it's 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, (eg. the fact that all roses are flowers). We will use Σ 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 LLFs are terms of type t.

3.2.3 An Example

Let's look at an example of how to translate a natural language sentence to an LLF. Specifically, we will 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 `Man` represents the property of being a man and $\lambda x.($ `Killed` x `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 $\forall_{o:Bird}$ (o 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 `Is` to be the top level operator and `The` to be applied to `Man` on the lowest level, instead, `The` is the top level operator and `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 `Who Man Walk` represents the property of being a man who walks.

To return to our example, we now have $\Sigma = \{The : (e \to t) \to (e \to t) \to t$, $Woman : e \to t$,

$Who : (e \to t) \to (e \to t) \to e \to t$, $Kill, Love : e \to e \to t$, $Bianca, Clark : e\}$. With this we can form the LLF `The (Who Woman (`λx` (Kill `x` Bianca)) (`λy` (Love `y` Clark)))` in the following manner:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\Sigma, y : e \vdash \texttt{Love: } e \to e \to t \qquad \Sigma, y : e \vdash y{:}e}{\Sigma, y : e \vdash \texttt{Love } y\texttt{: } e \to t} \qquad \Sigma, y : e \vdash \texttt{Clark: } e}{\Sigma, y : e \vdash \texttt{Love } y \texttt{ Clark: } t}}{\Sigma \vdash \lambda y \texttt{ (Love } y \texttt{ Clark): } e \to t}}{\Sigma \vdash \lambda y \texttt{ (Love } y \texttt{ Clark): } e \to t} \qquad \dfrac{\Sigma \vdash \texttt{The} : (e \to t) \to (e \to t) \to t \qquad \dfrac{\text{A}}{\Sigma \vdash \texttt{Who Woman (}\lambda x \texttt{ (Kill } x \texttt{ Bianca):} e \to t}}{\Sigma \vdash \texttt{The (Who Woman (}\lambda x \texttt{ (Kill } x \texttt{ Bianca)):} e \to t \to t)}}{\Sigma \vdash \texttt{The (Who Woman (}\lambda x \texttt{ (Kill } x \texttt{ Bianca)) (}\lambda y \texttt{ (Love } y \texttt{ Clark))):} t}$$

Where the subproof A is as follows:

A

$$\dfrac{\dfrac{\Sigma \vdash \texttt{Who:}(e \to t) \to (e \to t) \to e \to t \qquad \Sigma \vdash \texttt{Woman:} e \to t}{\Sigma \vdash \texttt{Who Woman:}(e \to t) \to e \to t} \qquad \dfrac{\dfrac{\dfrac{\Sigma, x : e \vdash \texttt{Kill: } e \to e \to t \qquad \Sigma, x : e \vdash x{:}e}{\Sigma, x : e \vdash \texttt{Kill } x\texttt{: } e \to t} \qquad \Sigma \vdash, x : e \texttt{ Bianca: } e}{\Sigma, x : e \vdash \texttt{Kill } x \texttt{ Bianca: } t}}{\Sigma \vdash \lambda x \texttt{ (Kill } x \texttt{ Bianca): } e \to t}}{\Sigma \vdash \texttt{Who Woman (}\lambda x \texttt{ (Kill } x \texttt{ Bianca):} e \to t}$$

3.2.4 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 (`The (Who Woman (`λx` (Love `x` Fiona)) (`λy` (Love `y` 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 `Who`, `Kill` and `Love` the sentence becomes `The (Woman ‘Who’ (`λx` (`x` ‘Kill’ Bianca)) (`λy` (`y` ‘Love’ Clark))` which is much closer to the original natural language sentence. We will place `‘ ’` 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 $e \rightarrow t$) and quantifiers (constants of type $(e \rightarrow t) \rightarrow (e \rightarrow t) \rightarrow t$) in the language. The tree represents a lexicographical ordering where every entity that has the child property also has the parent 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 (`Man ‘Who’ Walk`) $\subseteq$ `Man`.

Definition 1.
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

$$\frac{\Sigma \vdash x \subseteq y \qquad \Sigma \vdash y \subseteq z}{\Sigma \vdash x \subseteq z}$$

Reflexive Ordering

$$\frac{}{\Sigma \vdash x \subseteq x}$$

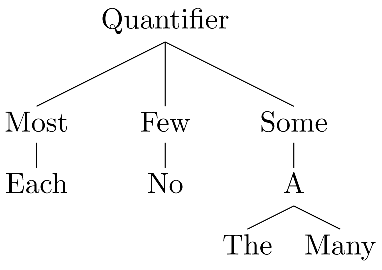
Who Ordering 1

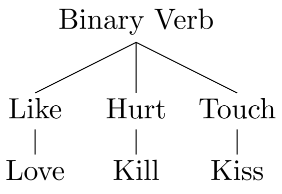
$$\frac{}{(x\ \texttt{`Who'}\ y)\subseteq x}\ \text{with } \Sigma \vdash x, y : e \to t$$

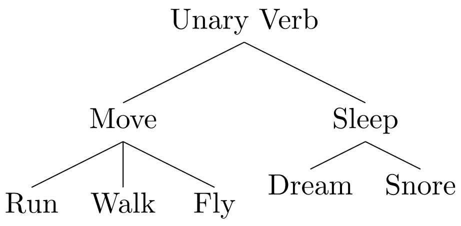
Who Ordering 2

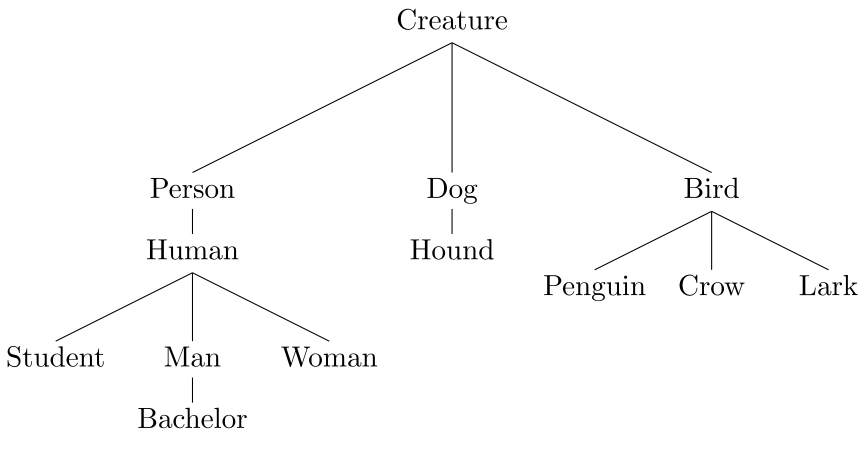
$$\frac{}{(x\ \texttt{`Who'}\ y)\subseteq y}\ \text{with } \Sigma \vdash x, y : e \to t$$

The trees that correspond to the subset of natural language that we are interested in are as follows[1] :









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

[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.

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 2.
The following rules define the *incompatibility relation*:

Base Restrictions

$$\frac{x \not\approx y \in \Sigma}{\Sigma \vdash x \not\approx y}$$

Symmetric Restrictions

$$\frac{\Sigma \vdash x \not\approx y}{\Sigma \vdash y \not\approx x} \text{ with } \Sigma \vdash x, y : e \to t$$

Restriction Ordering

$$\frac{\Sigma \vdash x' \subseteq x \qquad \Sigma \vdash x \not\approx y}{\Sigma \vdash x' \not\approx y} \text{ with } \Sigma \vdash x, y : e \to t$$

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`

3.3.3 Implementation

Both lexicographical ordering and lexicographical restrictions are implented using a combination of inductive types, for the ordering these are `LexEdge` and `Lex`, the first represents the edges of the tree while the second represents the fact that one property is an ancestor of another. Thus `Lex` $(x\ o)$ $(y\ o)$ models $x \subseteq y$ and `LexEdge` $(x\ o)$ $(y\ o)$ models that x is a child node of y in the lexicographic ordering.

Lexicographical restrictions uses `RestrictedCombination` and `InvalidCombination`. `InvalidCombination` just implements the above list of restrictions and `RestrictedCombination` uses that implementation and combines it with `Lex` to adhere to the Restriction Ordering requirement.

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 $q \in c$ so that we can derive that something is not true if it is not part of the context. Doing this naively would mean that every consistent context has infinite size. 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 quantifier statements. 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.

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.

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 `A` 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.

In the subset of natural language that we are modelling we have three ill defined quantifiers: `Many`, `Most` and `Few`

3.4.3 Formalization

Given all of that, we can now define the rules 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 3.
Formally a context c is *consistent* if the following rules as well as the rules in definition 4 hold:

Context Classification

$$\frac{p \in c}{\mathrm{Cont}(p)} \text{ with } \Sigma \vdash p : t$$

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

Who Derivation in Contexts

$$\frac{x\ \ o \in c \qquad y\ \ o \in c}{(x\ \texttt{'Who'}\ y)\ \ o \in c} \text{ with } \Sigma \vdash o : e,\ x, y : e \to t$$

$$\frac{(x\ \texttt{'Who'}\ y)\ o \in c}{x\ o \in c}\ \text{with } \Sigma \vdash o : e,\ x, y : e \to t$$

$$\frac{(x\ \texttt{'Who'}\ y)\ o \in c}{y\ o \in c}\ \text{with } \Sigma \vdash o : e,\ x, y : e \to t$$

Lexicographic Ordering in Contexts

$$\frac{\Sigma \vdash x \subseteq y \qquad x\ o \in c}{y\ o \in c}\ \text{with } \Sigma \vdash o : e,\ x, y : e \to t$$

Lexicographic Restrictions in Contexts

$$\frac{\Sigma \vdash x \not\approx y \qquad x\ o \in c}{\neg\ (y\ o \in c)}\ \text{with } \Sigma \vdash o : e,\ x, y : e \to t$$

Implementation

In the Coq implementation a context is simply a list of LLFs, it can theoretically hold any kind of LLF but we will only access it through the function `Contains` that filters out all LLFs but the three kinds mentioned above. Consistency is verified using an inductive type `Consistent` that verifies that it adheres to lexicographical knowledge and monotonicity (see below).

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, therefor 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, therefor 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 the 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.

Quantifier	First Argument	Second Argument
A	Up	Up
Each	Down	Up
No	Down	Down
The	None	Up
Many	Up	Up
Few	Down	Down
Most	None	Up

Definition 4.
The following rules extend the definition of a consistent context with *quantifier monotonicity*:

Upwards Monotonicity 1

$$\frac{z\ \ x\ \ y \in c \qquad \Sigma \vdash z_1^\uparrow \qquad \Sigma \vdash x \subseteq x'}{z\ \ x'\ \ y \in c}\ {\scriptstyle \Sigma\vdash x,x',y:e\to t;z:(e\to t)\to(e\to t)\to t}$$

Upwards Monotonicity 2

$$\frac{z\ \ x\ \ y \in c \qquad \Sigma \vdash z_2^\uparrow \qquad \Sigma \vdash y \subseteq y'}{z\ \ x\ \ y' \in c}\ {\scriptstyle \Sigma\vdash x,y,y':e\to t;z:(e\to t)\to(e\to t)\to t}$$

Downwards Monotonicity 1

$$\frac{z\ \ x'\ \ y \in c \qquad \Sigma \vdash z_1^{\downarrow} \qquad \Sigma \vdash x \subseteq x'}{z\ \ x\ \ y \in c}\ {\scriptstyle \Sigma\vdash x,x',y:e\to t;z:(e\to t)\to(e\to t)\to t}$$

Downwards Monotonicity 2

$$\frac{z\ \ x\ \ y' \in c \qquad \Sigma \vdash z_2^{\downarrow} \qquad \Sigma \vdash y \subseteq y'}{z\ \ x\ \ y \in c}\ {\scriptstyle \Sigma\vdash x,y,y':e\to t;z:(e\to t)\to(e\to t)\to t}$$

Quantifier Monotonicity

$$\frac{z\ \ x\ \ y \in c \qquad \Sigma \vdash z \subseteq z'}{z'\ \ x\ \ y \in c}\ {\scriptstyle \Sigma\vdash x,y:e\to t;zz':(e\to t)\to(e\to t)\to t}$$

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 `Not`. The first definition that springs to mind is something like:

Not Derivation

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

The problem is that Coq does not allow the negation of an 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 Σ. $c \not\models_\Sigma p$, pronounced *p is false in c*, on the other hand models that $\neg p$ can be derived from c given Σ (We will sometimes drop the Σ 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 4.2.

Mutual Inductive Types

Implementation wise the upshot from this is that we use two mutual inductive types `IsTrue` and `IsFalse` to implement $\models$ and $\not\models$. 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.

As a consequence, the resulting inductive type has quite a substantial number of constructors. Most of them are direct implementations of their given definitions, though, so this didn't lead to a dramatic increase in workload. There are some performance issues that result from all these constructors which is why we have attempted to keep the number of constructors to a minimum.

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 5.
The following rules provide a basic definition for the *is true in* relation and the *is false in* relations. This definition is extended in definitions 6, 7, 8, 9, 10, 11 and 12.

Basic Truth Derivation

$$\frac{p \in c}{c \models_\Sigma p} \text{ with } \Sigma \vdash p : t$$

Basic False Derivation

$$\frac{\texttt{Cont}(p) \qquad p \notin c}{c \not\models_\Sigma p} \text{ with } \Sigma \vdash p : t$$

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 definitions used in propositional logic.

Definition 6.
These rules extend definition 5 with the definitions for boolean operators.

And Derivation

$$\frac{c \models_\Sigma p \qquad c \models_\Sigma q}{c \models_\Sigma (p \texttt{ `And' } q)} \text{ with } \Sigma \vdash p, q : t$$

$$\frac{c \not\models_\Sigma p}{c \not\models_\Sigma (p \texttt{ `And' } q)} \text{ with } \Sigma \vdash p, q : t$$

$$\frac{c \not\models_\Sigma q}{c \not\models_\Sigma (p \texttt{ `And' } q)} \text{ with } \Sigma \vdash p, q : t$$

Or Derivation

$$\frac{c \models_\Sigma p}{c \models_\Sigma (p \texttt{ `Or' } q)} \text{ with } \Sigma \vdash p, q : t$$

$$\frac{c \models_\Sigma q}{c \models_\Sigma (p \texttt{ `Or' } q)} \text{ with } \Sigma \vdash p, q : t$$

$$\frac{c \not\models_\Sigma p \qquad c \not\models_\Sigma q}{c \not\models_\Sigma (p \texttt{ `Or' } q)} \text{ with } \Sigma \vdash p, q : t$$

Not Derivation

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

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

Implies Derivation

$$\frac{c \models_\Sigma q}{c \models_\Sigma \texttt{(}p \texttt{ `Implies' } q\texttt{)}} \text{ with } \Sigma \vdash p, q : t$$

$$\frac{c \models_\Sigma \texttt{Not } p}{c \models_\Sigma \texttt{(}p \texttt{ `Implies' } q\texttt{)}} \text{ with } \Sigma \vdash p, q : t$$

$$\frac{c \models_\Sigma p \qquad c \not\models_\Sigma q}{c \not\models_\Sigma \texttt{(}p \texttt{ `Implies' } q\texttt{)}} \text{ with } \Sigma \vdash p, q : t$$

Implementation

The implementations of the boolean operators are mostly straighforward translations of the rules associated with them. We did try to minimize the number of necessary constructors by combining axioms that implement a disjunction. So both of the positive axioms for `Or` are implemented by a single constructor:

$$\texttt{forall } c\ p\ q\texttt{, Consistent } c \implies \texttt{IsTrue } c\ p \lor \texttt{IsTrue } c\ q \implies \texttt{IsTrue } c\ \texttt{(}p \texttt{ `Or' } q\texttt{)}$$

The same thing was done for `Implies` and the negation of `And`.

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.

A

We will consider `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 `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, `A Woman Walk` and `A Walk Woman`[2] have the same truth condition.

Translating this meaning into axioms is fairly straightforward, we have an axiom for the positive version which states that $\texttt{A}\ x\ \ y$ 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 $\texttt{A}\ x\ \ y$ is false if for every o either x o or $y\ \ o$ fails to hold.

Definition 7.
The following rules extend definition 5 with a definition for `A`.

A Derivation

$$\frac{c \models_\Sigma x\ \ o \qquad c \models_\Sigma y\ \ o}{c \models_\Sigma \texttt{A}\ x\ \ y}\ \text{with } \Sigma \vdash o : e, x, y : e \rightarrow t$$

$$\frac{\forall_{o:e}\ (c \not\models_\Sigma((x\ \ o\ \ \texttt{'And'}\ \ y\ \ o))}{c \not\models_\Sigma \texttt{A}\ x\ \ y}\ \text{with } \Sigma \vdash x, y : e \rightarrow t$$

[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'.

Some

A number of test cases from the Prolog test suite (see section 4.2.1) use the word `Some`. The semantic content of `Some` is identical to `A`, thus they will be considered to be synonymous.

Definition 8.
The following rules extend definition 5 with a definition for `Some`.

Some Derivation

$$\frac{c \models_\Sigma \texttt{Some}\ x\ y}{c \models_\Sigma \texttt{A}\ x\ y}\ \text{with } \Sigma \vdash x, y : e \to t$$

$$\frac{c \not\models_\Sigma \texttt{Some}\ x\ y}{c \not\models_\Sigma \texttt{A}\ x\ y}\ \text{with } \Sigma \vdash x, y : e \to t$$

Each

If `A` is the natural language equivalent of the existential quantifier then `Each` is the equivalent of the universal quantifier. Unlike with `A` the arguments of `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 `Each Crow Bird` and `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, `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 `Each` x y is also false.

Definition 9.
The following rules extend definition 5 with a definition for `Each`.

Each Derivation

$$\frac{\forall_{o:e}\ (c \models_\Sigma \texttt{(}x\ o\ \texttt{`Implies'}\ y\ o\texttt{))})}{c \models_\Sigma \texttt{Each}\ x\ y}\ \text{with } \Sigma \vdash x, y : e \to t$$

$$\frac{\exists_{o:e}\ (c \models_\Sigma \texttt{(}x\ o\ \texttt{`And'}\ \texttt{(Not}\ \texttt{(}y\ o\texttt{)))})}{c \not\models_\Sigma \texttt{Each}\ x\ y}\ \text{with } \Sigma \vdash x, y : e \to t$$

The

The word `The` is slightly trickier because its meaning in natural language is somewhat vague. We will use the common interpretation that `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, `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 10.
The following rules extend definition 5 with a definition for `The`.

The Derivation

$$\frac{c \models_\Sigma \ (x\ \ o\ \ \texttt{`And'}\ \ y\ \ o) \qquad \forall_{o':e}\ (c \models_\Sigma \ (x\ \ o'\ \ \texttt{`And'}\ \ y\ \ o') \implies (o' = o))}{c \models_\Sigma \ \texttt{The}\ \ x\ \ y} \text{ with } \Sigma \vdash o : e, x, y : e \to t$$

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

$$\frac{c \models_\Sigma x\ \ o\ \ \texttt{`And'}\ \ y\ \ o \qquad c \models_\Sigma x\ \ o'\ \ \texttt{`And'}\ \ y\ \ o' \qquad o \neq o'}{c \not\models_\Sigma \texttt{The}\ \ x\ \ y} \text{ with } \Sigma \vdash o, o' : e, x, y : e \to t$$

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 11.
The following rules extend definition 5 with a definition for `No`.

No Derivation

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

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

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.

Implementation

There are number of minor ways in which the implementation deviates from the specifications given in this section. First, whenever possible we have again combined rules to reduce the number of necessary constructors. Second, we require that quantifier arguments applied to objects 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. Third and somewhat related we skip a step in verifying that such statements are true, we simply check whether they are part of the context directly, rather than first checking whether they are true, then determining that simple statements and who statements can only be true if they are part of the context and then checking whether they are part of the context. Finally, since `Some` is synonymous with `A` we did not give it a separate definition, instead defining it as a synonym directly.

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 ??. 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 Truthfull About'`, `'Is Lying About'`, `'Is Right About'` and `'Is Wrong About'` are defined in this way.

Definition 12.
The following rules extend definition 5 with a definitions for belief and claim statements.

Is Right About Derivation

$$\frac{c \models_\Sigma p \qquad c \models_\Sigma o\ \texttt{`Believe'}\ p}{c \models_\Sigma o\ \texttt{`Is Right About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \not\models_\Sigma p}{c \not\models_\Sigma o\ \texttt{`Is Right About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \not\models_\Sigma o\ \texttt{`Believe'}\ p}{c \not\models_\Sigma o\ \texttt{`Is Right About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

Is Wrong About Derivation

$$\frac{c \not\models_\Sigma p \qquad c \models_\Sigma o\ \texttt{`Believe'}\ p}{c \models_\Sigma o\ \texttt{`Is Wrong About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \models_\Sigma p}{c \not\models_\Sigma o\ \texttt{`Is Wrong About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \not\models_\Sigma o\ \texttt{`Believe'}\ p}{c \not\models_\Sigma o\ \texttt{`Is Wrong About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

Is Truthful About Derivation

[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.

$$\frac{c \models_\Sigma o\ \texttt{`Say'}\ p \qquad c \models_\Sigma o\ \texttt{`Believe'}\ p}{c \models_\Sigma o\ \texttt{`Is Truthful About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \not\models_\Sigma o\ \texttt{`Say'}\ p}{c \not\models_\Sigma o\ \texttt{`Is Truthful About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \not\models_\Sigma o\ \texttt{`Believe'}\ p}{c \not\models_\Sigma o\ \texttt{`Is Truthful About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

Is Lying About Derivation

$$\frac{c \models_\Sigma o\ \texttt{`Say'}\ p \qquad c \not\models_\Sigma o\ \texttt{`Believe'}\ p}{c \models_\Sigma o\ \texttt{`Is Lying About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \not\models_\Sigma o\ \texttt{`Say'}\ p}{c \not\models_\Sigma o\ \texttt{`Is Lying About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

$$\frac{c \models_\Sigma o\ \texttt{`Believe'}\ p}{c \not\models_\Sigma o\ \texttt{`Is Lying About'}\ p}\ {\scriptstyle \Sigma\vdash o:e,p:t}$$

Originally we also tried to implement more global variants of these terms. The term `‘Is Truthfull’`, 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.

3.6 LTac Commands

Alongside the model, several Ltac tactics were implemented to facilitate writing proofs concerning LLFs, we will go over the most prominent ones here.

3.6.1 Knowledge

The way the model is set up means that a lot of 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.

3.6.2 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 `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 4.2), it states:

$$\texttt{forall}\ c\ p\ q\texttt{,}\ \texttt{Consistent}\ c \implies \texttt{SimpleMonotonic}\ p\ q \implies \texttt{Contains}\ c\ p \implies \texttt{Contains}\ c\ q.$$

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: IsTrue _ (A ?P ?Q) |− IsTrue _ (A _ ?Q)] ⟹ apply L_MonotonicityA1 with (x1 := P);
                                                             Knowledge
      | [ H: IsTrue _ (A ?P ?Q) |− IsTrue _ (A ?P _)] ⟹ apply L_MonotonicityA2 with (y1 := Q);
                                                             Knowledge
      | [ H: IsTrue ?c (A ?P _) |− IsTrue ?c (A _ ?Q)] ⟹ assert (IsTrue c (A P Q));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.

3.6.3 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 Man Walk) And (A Woman Walk))` becomes `IsTrue` c `(A Man Walk)` $\wedge$ `IsTrue` c `(A Woman Walk)` and IsTrue c (A Man Walks) splits into `Simple (Man` o`)` $\vee$ `IsWho (Man` o`)`, `Simple (Walk` o`)` $\vee$ `IsWho (Walk` o`)`, `Contains` c `(Woman` o`)` and `Contains` c `(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`[4] 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 it's 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]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.

3.6.4 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.

Chapter 4

Verification

4.1 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.

4.2 Lemmas

The first verification step was to establish a number of lemmas that we knew the model should adhere to. These lemmas are the basic properties that we expect from a system like this (eg. the fact that every statement is either true or false and not both) as well as things like monotonicity for well defined quantifiers (which follows from their definitions rather than being hardcoded in).

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).

4.2.1 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 way the prolog prover works is that it takes two LLFs and 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.

4.2.2 Testscenarios

As a final verification step we wrote a number of test scenarios of our own, we constructed these test scenarios from several issues that came up while testing the system. Once such an issue was established a test was generated that should only be provable once the issue had been fixed. To this list we added a number of tests that amongst them contained all the constants for which we have implementation in an effort to find issues that didn't show up in basic testing. Most of these test now can be solved by a singe application of `ConvertAll` plus a few bookkeeping tactics.

[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.

Chapter 5

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.

5.1 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 therefor a bird flies) and from an ill defined quantifiers to a well defined quantifier (Many birds fly therefor 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 therefor most birds fly) and between ill defined quantifiers (theorectically 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.

5.2 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 that require additional user input. We would have liked to fix this.

5.3 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.

Bibliography

[1] Yves Bertot. *Coq in a Hurry.* April 2010.

[2] Yves Bertot and Pierre Castran. *Coq'Art: The Calculus of Inductive Constructions.* Texts in Theoretical Computer Science. An EATCS Series. April 2004.

[3] The coq proof assistant. `http://coq.inria.fr/`. Accessed: 2013-07-31.

[4] Gottlob Frege. *Begriffsschrift, eine der arithmetischen nachgebildeten Formelsprache des reinen Denkens.* Halle: Nebert, 1879.

[5] Rutger Huijben. *A Coq module for natural logic, comprehensive version.* 2013.

[6] Richard Montague. *Universal grammar.* Theoria, 36, 1970.

[7] Reinhard Muskens. An analytic tableau system for natural logic. In M. Aloni, H. Bastiaanse, T. de Jager, and K. Schulz, editors, *Logic, Language and Meaning*, volume 6042 of *Lecture Notes in Artificial Intelligence*, pages 104–113. Springer, 2010.

[8] Aarne Ranta. Grammatical framework: A type-theoretical grammar formalism. In *Journal of Functional Programming*, volume 14, pages 145–189. 2004.

[9] Christian Retore. Variable types for meaning assembly: a logical syntax for generic noun phrases introduced by 'most'. In *Recherches linguistiques de Vincennes*, volume 41, pages 83–102. 2012.