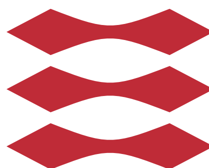
A Logical Approach to Anaphora in Natural Language

Michael Sejr Schlichtkrull



Kongens Lyngby 2013
B.Sc.-2013-21

Technical University of Denmark
DTU Compute
Building 303B, Matematiktorvet, DK-2800 Kongens Lyngby, Denmark
Phone +45 45253351, Fax +45 45882673
reception@compute.dtu.dk
www.compute.dtu.dk B.Sc.-2013-21

Summary (English)

Through a synthesis between Discourse Representation Theory and Montague Semantics, this thesis presents a formal logical approach to anaphora resolution in natural languages based on the strategy described by Reinhard Muskens in the 1991 paper Anaphora and the Logic of Change.

A method for validating co-indexings after translation by employing a set of filters is proposed, and the approach is illustrated through filters based on subtypes, logic, and an adaptation of Chomsky's Government and Binding theory.

While the approach is shown to be capable of standing on its own, the proposed design should be seen as one half of a system that utilizes both logical and statistical means.

The thesis is accompanied by a *proof of concept* implementation, demonstrating the described strategy both as translator and as anaphora resolver.

Summary (Danish)

Gennem en syntese af diskursrepresentationsteori og Montague semantic præsenterer denne afhandling en formel logisk tilgang til anaforisk resolution i naturligt sprog baseret på strategien beskrevet af Reinhard Muskens i artiklen *Anaphora and the Logic of Change* fra 1991.

Der fremsættes en metode til at validere co-indekseringer efter oversættelse gennem anvendelse af filtre. Tilgangen illustreres med filtre baseret på deltyper, logisk struktur og en tilpasning af Chomsky's Government and Binding teori.

Selvom det vises at proceduren er i stand til at stå alene som oversættelsesproces, skal det foreslåede design ses som en del af et system der benytter både logiske og statistiske metoder.

Afhandlingen indeholder også en implementering af et *konceptbevis*, som demonstrerer den beskrevne strategi både som oversættelsesproces og som proces til anaforisk resolution.

Preface

This thesis was prepared at the Department of Applied Mathematics and Computer Science at the Technical University of Denmark in fulfilment of the requirements for acquiring a B.Sc. in Software Technology.

The thesis deals with the resolution of anaphora in natural language, and the translation of natural language to a logical representation. A solution based on a formal logical approach is presented. The reader is assumed to have a reasonable understanding of combinatory logic and formal languages, as well as a rudimentary knowledge of the theory of grammar.

The project was conducted in the period from the 4th of February to the 1st of July 2013 under the supervision of Jørgen Villadsen. The project was valued at 15 ECTS points.

Lyngby, 01-July-2013

Michael Sejr Schlichtkrull

Michael Sejr Schlichtkrull

Acknowledgements

I would like to thank my supervisor Jørgen Villadsen for his outstanding guidance throughout the process, and for his invaluable assistance in the search for the literature which made this thesis possible.

Moreover, I would like to recognize Jørgen's course Logical Systems and Logic Programming (02156) along with the courses Functional Programming (02157) and Computer Science Modelling (02141), each of which have provided me with insights valuable to this thesis.

Contents

Summary (English) **i**

Summary (Danish) **iii**

Preface **v**

Acknowledgements **vii**

1 Introduction **1**
1.1 Key Concepts 2
1.2 Generation of Indexings 3

2 Logical Background **5**
2.1 Typed Lambda Calculus 6
2.2 Dynamic Logic 8
2.3 A Logical Synthesis 9

3 Categorial Grammar **11**
3.1 Dictionary 13
3.2 The Problem of Causality 16
3.3 Encyclopaedic Semantics 18
3.4 Defining the Semantic Function 20

4 Reduction and Simplification **21**
4.1 From Meaning to Truth 21
4.2 Anaphora as Generators of Entities 22
4.3 From States to Entities 23

5 Filtering of Indexings **25**
5.1 Syntactic Filtering . . . 25
5.2 Gender-based Filtering . . . 27
5.3 Nominal Filtering . . . 28

6 Implementation **29**
6.1 Data Structures . . . 30
6.2 Framework . . . 31
6.3 Tokenization . . . 32
6.4 Parsing . . . 33
6.5 Beta-Reduction . . . 34
6.6 Simplification & Unselective Binding Lemma . . . 35
6.7 Validation of Indexings . . . 40

7 Examples **43**
7.1 A Detailed Example . . . 43
7.2 Other Interesting Cases . . . 46

8 Discussion **49**
8.1 Syntactic Ambiguities . . . 50
8.2 Logical Filtering . . . 50
8.3 A Little Help From Statistics . . . 52
8.4 Other Improvements . . . 52

9 Conclusion **55**

A Source Code **57**

B Auxiliary Code **85**

C Sample Dictionary **87**

D Test Suite **93**

Bibliography **99**

Chapter 1

Introduction

When humans communicate, we constantly use expressions that contain *references* to objects instead of merely acting as *descriptions* of objects - anaphora. We expect common understanding of such lingual signs to form a basis for decoding communication. Indeed, a conversation between humans devoid of internal references appears almost unnatural, as we intuitively bind syntactic elements together through reference. With that in mind, it is not surprising that the study of anaphoric constructions since the earliest attempts to formalize language has been an essential topic among philosophers, linguists, and logicians alike.

In Anaphora and the Logic of Change [Mus91], a strategy is proposed for translating and resolving sentences with anaphoric ambiguities through a hybrid of Montague semantic and Discourse Representation Theory. Muskens' approach is an attempt to bridge the gap, combining the successful treatment of anaphora in DRT with the elegant semantics of Montague. In this thesis, I shall explore the opportunities presented by that approach, with additional consideration towards deixis and encyclopaedic semantics.

To properly translate a full sentence, two things are necessary: An indexing and a translation process. These represent respectively an assignment of objects to references and a correspondence between the indexed sentence and a logical expression.

Depending on the indexing used to translate a sentence, the meaning of the sentence may change dramatically. As such, it is important to ensure that any indexing to be translated is sound. In this paper, I shall combine the translation strategy devised by Muskens with a set of *filters*, guaranteeing that a sensible indexing is used. I have chosen three such constraints, each illustrating a different approach to filtering: Syntactic, nominal and gender-based.

To demonstrate the proposed method of validation and translation, a *proof of concept* implementation has been constructed. Furthermore, the program has been combined with a primitive indexing generator to show how a full translation engine can emerge.

1.1 Key Concepts

Any language fragment consists of a sequence of symbols, to which meaning are assigned. Such symbols are referred to as words. The set of interpretable words is denoted W. A sequence of words is referred to as a **sentence**.

A **syntax** describes a set of sentences referred to as **well-formed**. Typically, syntax is defined on the basis of a set of rules called a **grammar**. Most syntaxes describing natural languages divide words into subsets called **syntactic classes** - for example verbs, nouns, and adjectives. The **semantics** of a sentence denotes the meaning behind the text, expressed as a logical statement. I will use the notation $[\![s]\!]$ to denote the semantics of a sentence s.

In most cases, discourse concerns itself with objects in the real world - people, places, items. I use the term **entity** to describe such an object. When encountered in natural language, entities are referred to by nouns, pronouns, or proper nouns. A single word referring to an entity is denoted a **reference**, while the entity referred to is called a **referent**. The relationship between reference and referent is similar to that described in the beginning of the 20th century by Saussure, distinguishing between image (*signifiant*) and concept (*signifié*). [Sau16]

When two references share a referent, they are said to **corefer**. To denote that two references have the same referent, a subscript numbering is used:

$$\text{"}(Bob)_1 \text{ owns (a donkey)}_2\text{. (He)}_1 \text{ beats (it)}_2\text{."}$$

Such an assignment of referents to references is denoted a **coindexing** (or just an **indexing**). Mostly, one specific reference can be said to introduce each referent to the discourse - in the example, these are respectively *(Bob)* and *(A donkey)*. This reference is said to **generate** the entity. If a reference occurs later in the text than the generating reference, it is called an **anaphor**. If it occurs earlier in the text, it is called a **cataphor**.

In the semantics used by Muskens [Mus91], the indexing is associated with the article rather than with the noun: "*(Bob)*$_1$ *owns (a)*$_2$ *donkey*. Intuitively, this makes sense as the difference between a generating reference (*a donkey*) and a non-generating reference (*the donkey*) often lies only in the article.

In corpora, entities may describe illusive concepts such as actions (*"Alice did it"*) or ideas (*"Bob thought so"*). For the purpose of this thesis, entities are limited to denote objects mentionable by a noun or proper noun. Furthermore, the study of cataphora is beyond the scope of this paper.

1.2 Generation of Indexings

As the main objective of the project is to translate and validate rather than generate indexings, the program is limited to the use of a primitive generation strategy as described in the following paragraphs.

Generating references are given a unique number, while non-generating references are associated with a variable $1 \leq x_i \leq n$, where n is the total number of references. In the case of the previous example, this indexing corresponds to the following:

$$\text{"(Bob)}_1 \text{ owns (a)}_2 \text{ donkey. (He)}_{x_1} \text{ beats (it)}_{x_2}\text{."}$$

Based on this, a sequence of all possible indexings is generated. In the case of the example, the sequence consists of $(1,2,1,1)$, $(1,2,1,2)$,$(1,2,1,3)$, $(1,2,1,4)$, $(1,2,2,1)$, $(1,2,2,2)$, $(1,2,2,3)$, $(1,2,2,4)$, $(1,2,3,1)$, $(1,2,3,2)$, $(1,2,3,3)$, $(1,2,3,4)$, $(1,2,4,1)$, $(1,2,4,2)$, $(1,2,4,3)$, and $(1,2,4,4)$.

The sentence is translated using each indexing until one is validated by the filters.

Chapter 2

Logical Background

The semantics of the proposed approach are based on two logical systems: Typed lambda calculus and dynamic logic. In this chapter I will introduce the two logics - in section 2.1 and section 2.2, respectively. I will use the definition of typed λ-calculus employed by Curry rather than the one by Church [Bar92].

In section 2.3 I shall introduce the synthesis between the two logics presented by Muskens along with the concept of stores essential to his approach to anaphora resolution. [Mus91]

For a more thorough analysis of λ-calculus and dynamic logic, see respectively Barendregt [Bar92] and Harel et al [HKT84].

2.1 Typed Lambda Calculus

As in ordinary lambda-calculus, terms are defined inductively through three rules. Let x be a variable or constant. Furthermore, let s and t be λ-terms. Then, the following are λ-terms:

Variable:	x
Abstraction:	$\lambda x.t$
Application:	(ts)

A type is defined as a set of λ-terms abiding to a set of rules called **typing rules**. The shorthand notation $t : \sigma$ is used to denote $t \in \sigma$. The set of types is inductively constructed from a set of primitive types Φ and a type constructor $\rightarrow$. As such, if τ and σ are types, $\tau \rightarrow \sigma$ is a type as well.

The typing rules are defined inductively on the basis of a set Γ of initial assumptions, which assign types to some given terms. Thus, Γ is a set of pairs (t, σ) where t is a term and σ is a type. The typing rules are as follows, where M and N denote arbitrary terms and x an arbitrary variable:

Axiom:	$\dfrac{(M, \sigma) \in \Gamma}{\Gamma \vdash M : \sigma}$
$\rightarrow$-introduction:	$\dfrac{\Gamma \vdash x : \sigma \qquad \Gamma \vdash M : \tau}{\Gamma \vdash \lambda x.M : (\sigma \rightarrow \tau)}$
$\rightarrow$-elimination:	$\dfrac{\Gamma \vdash M : (\sigma \rightarrow \tau) \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash (MN) : \tau}$

Remark 2.1. *If it holds for all terms x such that $\exists y (x, y) \in \Gamma$ that x is a variable and that $\neg \exists y_1 y_2 (x, y_1) \in \Gamma \wedge (x, y_2) \in \Gamma \wedge y_1 \neq y_2$, then the types are disjunct. [Bar92] Less formally, the types defined through the typing rules are disjunct if only variables are initially assigned types, and if every variable is only assigned one type.*

All terms for which a type can be derived using these rules are said to be *well-typed*. For the purposes of this paper, only well-typed terms are of interest. Furthermore, the conditions described in remark 2.1 hold. Therefore, every term belongs to a single type, and the notation $\tau(M)$ can be used to refer to the type of a term M.

Terms in typed and untyped λ-calculus alike may be reduced through the scheme of α-conversion and β-reduction. Using the notation $t[x/y]$ to signify the substitution of x for y in t, the rules are as follows:

$$\begin{array}{ll} \alpha\text{-conversion:} & \lambda x.t_1 \equiv \lambda y.(t_1[x/y]) \\ \beta\text{-reduction:} & (\lambda x.t_1)t_2 \equiv t_1[x/t_2] \end{array}$$

For reasons explained by Muskens [Mus91, p. 9], a three-typed lambda-calculus with primitive types $\Phi = \{e, s, t\}$ will be used. The three types represent respectively entities, states, and truth values.

The calculus can be extended to include the full first-order logic. A set of constants Ξ is introduced. Predicates, quantifiers and logical combinators are represented as constants in Ξ, using the following types:

Constant	Type
n-ary predicate	$e_1 \to ... \to e_n \to t$
$\vee$	$t \to (t \to t)$
$\wedge$	$t \to (t \to t)$
$\to$	$t \to (t \to t)$
$\neg$	$t \to t$
$\forall$	$s \to (t \to t)$ or $e \to (t \to t)$
$\exists$	$s \to (t \to t)$ or $e \to (t \to t)$
$=$	$s \to (s \to t)$ or $e \to (e \to t)$

Three subsets of this logic will be used: Λ, PL_s, and PL_e.

Definition 2.1. Λ *denotes the full set of well-typed terms.* PL_s *denotes the subset of* Λ *excluding any formula produced through abstraction.* PL_e *denotes the two-typed subset of* PL_s *excluding* s*.*

The reader may notice that PL_e is essentially ordinary predicate logic, while PL_s is predicate logic extended with a primitive type corresponding to states.

2.2 Dynamic Logic

Dynamic Logic is an extension of modal logic designed for reasoning about computation. As the name implies, dynamic logic concerns itself with situations in which the truth values of formulae change. This phenomenon is formalized through constructs called **programs**.

A program should be envisioned as an automata defining a transition between **states**, representing valuations of variables. For reasons outlined by Muskens [Mus91], programs should operate non-deterministically.

Programs operate on **formulae** - a set of logical expressions equivalent to PL_e. Importantly, the value of a formula depends on the values of the variables in the formula, and thus on the state.

The set of programs is defined inductively. Let v be a variable, let u be a variable or a constant, and let A be a formula. Then, two atomic programs are used by Muskens [Mus91]:

$v := u$	Assignment
$A?$	Test

Ordinarily, many combinators exist for creating compound programs. However, only two are used[Mus91]. Let γ and δ be programs. Then, the following are compound programs:

$(\gamma; \delta)$	Sequential Composition
$(\gamma \cup \delta)$	Non-deterministic Choice

A well-formed term of dynamic logic has the form $[P]F$, where, P is a program and F is a formula. $[P]F$ should be interpreted as the assertion that F holds immediately after executing P. In the context of natural languages, this notation is of little importance, as the goal is to interpret sentences as programs - not as full terms.

2.3 A Logical Synthesis

Representing dynamic logic in the language of Montague semantics requires a translation into typed λ-calculus. In regular dynamic logic, states are treated as functions from variables to values. As the reader may have noticed, this contradicts the notion of states as one of the primitive types introduced in section 2.1. The problem is solved by extending Λ with a set of constants $\{v_1, ..., v_n\}$ of the type $s \to e$. Following the terminology of Muskens, such functions are denoted **stores**.

Pertaining to stores, the notation $i[v]j$ is introduced as an abbreviation of $\forall u_{s\to e}((ST\ \ u \wedge u \neq v) \to uj = ui)$ [Mus91]. Here, ST is a function with the interpretation "is a store". Intuitively, the meaning of $i[v]j$ is that the state i and j agree on the values of all stores except possibly v.

A translation function Θ can now be defined. Formulae represent functions from states to truth values, and as such their translations should have type $s \to t$. Of particular importance is the translation of variables:

$$\Theta(v) = \lambda i(v\ i)$$

Constants have the same value regardless of the state. As such, the following translation is used:

$$\Theta(v) = \lambda i(v)$$

For compound expressions with combinator $\bullet$, the translation is:

$$\Theta(e_1 \bullet e_2) = \lambda i(\Theta(e_1)i \bullet \Theta(e_2)i)$$

The translations for quantifiers and negations is similar.

Programs represent relations between states. As such, they correspond to expressions of the type $s \to (s \to t)$. I will use the shorthand σ to cover this type. With that in mind, the following set of translations are defined:

$$\begin{aligned} \Theta(v := u) &= \lambda ij(i[v]j \wedge vj = ui) \\ \Theta(A?) &= \lambda ij(\Theta(A)i \wedge j = i) \\ \Theta(\gamma; \delta) &= \lambda ij \exists k(\Theta(\gamma)ik \wedge \Theta(\delta)kj) \\ \Theta(\gamma \cup \delta) &= \lambda ij(\Theta(\gamma)ij \vee \Theta(\delta)ij) \end{aligned}$$

Consider the example *he₁ sleeps*. Intuitively, the sentence should be understood as a test, verifying that the entity in store 1 is asleep. As such:

$$[\![he_1\ \ sleeps]\!] = \Theta(sleeps\ v_1?)$$

Translating into type theory, we get:

$$\begin{aligned}\Theta(\textit{sleeps}\ v_1?) &= \lambda i \lambda j(\Theta(\textit{sleeps}\ v_1)i \wedge i = j)\\ &= \lambda i \lambda j((\Theta(\textit{sleeps})i\ \Theta(v_1)i) \wedge i = j)\\ &= \lambda i \lambda j((\textit{sleeps}\ (v_1 i)) \wedge i = j)\end{aligned}$$

Generating references are treated as assignments introducing a new variable. As such, for the sentence $(a\ man)_1\, sleeps$:

$$\begin{aligned}[\![(a\ man)_1\ \ sleeps]\!] &= \Theta(((v_1 := x); (man\ v_1?)); (sleeps\ v_1?))\\ &= \lambda ij \exists k(\Theta((v_1 := x); (man\ v_1?))ik \wedge \Theta(sleeps\ v_1?)kj)\\ &= \lambda ij \exists kh(\Theta(v_1 := x)ih \wedge \Theta(man\ v_1?)hk \wedge \Theta(sleeps\ v_1?)kj)\\ &= \lambda ij \exists kh((i[v_1]h \wedge v_1 j = xi) \wedge \Theta(man\ v_1?)hk \wedge \Theta(sleeps\ v_1?)kj)\end{aligned}$$

Since x is a completely new entity, the statement $v_1 j = xi$ is tautological. As such:

$$= \lambda ij \exists kh(i[v_1]h \wedge \Theta(man\ v_1?)hk \wedge \Theta(sleeps\ v_1?)kj)$$

Finally, the tests can be translated through the now familiar process:

$$\begin{aligned}&= \lambda ij \exists kh(i[v_1]h \wedge man\ (v_1 h) \wedge h = k \wedge sleeps\ (v_1 k) \wedge k = j)\\ &= \lambda ij(i[v_1]j \wedge man\ (v_1 j) \wedge sleeps\ (v_1 j))\end{aligned}$$

In the next chapter, I shall demonstrate how Θ can be emulated by combining the semantics of individual words.

Chapter 3

Categorial Grammar

A categorial grammar is defined as a pair (P, C), where P is a set of elements called primitive categories and C is a set of elements called combinators. The set of categorial grammars is denoted Ω. Given a grammar $\gamma = (P, C)$, the language Γ of that grammar is defined inductively:

Axiom: $$\frac{p \in P}{p \in \Gamma}$$

Composition: $$\frac{c_1 \in \Gamma \qquad c_2 \in \Gamma \qquad \bullet \in C}{c_1 \bullet c_2 \in \Gamma}$$

For the purposes of this paper, only a subset of Ω denoted A/B-grammars [MR12] are relevant. The set of A/B-grammars is the set $\{\gamma \in \Omega | \gamma = (P, \{/, \backslash\})\}$, where P is an arbitrary set of primitive categories.

Note that for more complicated languages, there are sentences which cannot be represented by A/B-grammars. For the full English language, the introduction of additional combinators is necessary. [MR12]

For the purpose of derivation, a categorial grammar may be thought of as a sequent calculus. The category of a sequence is recursively derived through a deduction scheme. The following derivation rules are used in the case of A/B-grammars:

Axiom:	$\dfrac{}{A \vdash A}$
/-derivation:	$\dfrac{\Sigma \vdash A/B \qquad \Pi \vdash B}{\Sigma, \Pi \vdash A}$
\\-derivation:	$\dfrac{\Pi \vdash B\backslash A \qquad \Sigma \vdash B}{\Sigma, \Pi \vdash A}$

This reflects the intuitive notion of A/B and $B\backslash A$ as functions taking a category B as argument and yielding a category A. However, A/B takes the argument on the right side, while $B\backslash A$ takes the argument on the left side.

Remark 3.1. *In \-derivation, the ordering of* Σ *and* Π *is reversed in the conclusion. This is in accordance with the mentioned intuition and prepares for the equivalence of both derivations to the type constructor* $\rightarrow$*, as discussed in section 3.1.*

A sequence Σ is **derivable** iff there exists a category ς such that $\Sigma \vdash \varsigma$. Proofs are represented using a binary tree notation as seen below. A proof of a Σ is therefore referred to as a **derivation tree** of Σ. To illustrate the notation, consider the proof of $(A/B)/C, C, D, D\backslash B \vdash A$:

$$\dfrac{\dfrac{(A/B)/C \qquad C}{A/B} \qquad \dfrac{D\backslash B \qquad D}{B}}{A}$$

Notice the ordering of D and $D\backslash B$ conforming to remark 3.1.

For reasons outlined by Muskens [Mus91], I will use an A/B grammar with two primitives $P = \{S, E\}$. This grammar is denoted as γ, while the language of γ is denoted Γ.

In addition to the primitive categories, liberal use will be made of the following abbreviations:

$$\begin{aligned} VP &= S/E \\ N &= S\backslash E \\ NP &= S/(S/E) \end{aligned}$$

The abbreviations correspond to the syntactic classes of *Verb Phrases*, *Nouns* and *Noun Phrases*.

3.1 Dictionary

A dictionary is defined as a function $\Delta : W \to \Gamma \times \Lambda$ (see the previous section and definition 2.1, respectively). Note that if the full English language were to be modelled, it would be necessary to map each word to a set of pairs instead of a single pair, in order to accommodate for words with multiple meanings.

Given a sentence s, let Σ be the sequence of categories Δ associates with each word in s. A derivation tree $\Psi(s)$ is defined as a derivation tree of Σ. Furthermore, s is a well-formed sentence iff $\Sigma \vdash S$. For an example, consider the sentence s_1 = *a man sleeps*. In this case, assume that $\Sigma = (NP/N, N, VP)$. $\Psi(s_1)$ can then be constructed as:

$$\dfrac{\dfrac{\dfrac{a}{NP/N} \quad \dfrac{man}{N}}{NP} \qquad \dfrac{sleeps}{VP}}{S}$$

We see that the sentence is well-formed, as S is derived.

Now, the semantics function can be defined inductively. Let $P : \Gamma \to \Lambda$ be the projection $P(g, l) = l$. Then for a single word w:

$$[\![w]\!] \equiv P \circ \Delta(w)$$

Consider a sentence s. If s is derivable, there exist sequences of words Σ and Π such that $s = \Sigma, \Pi$, and either $\Sigma \vdash A/B \wedge \Pi \vdash B$ or $\Sigma \vdash B \wedge \Pi \vdash B\backslash A$. Essentially, the last derivation required to prove s. Then:

$$[\![s]\!] \equiv [\![\Sigma]\!][\![\Pi]\!]$$

If s is not derivable, $[\![s]\!]$ is left undefined. Since Σ and Π are trivially derivable, $[\![s]\!]$ is still defined for all derivable sentences.

As discovered by Montague [Mon70], there is an intimate connection between syntactic proofs and semantics. Reading the compositions / and \ as equivalent to the type constructor $\rightarrow$, a morphism ϕ from syntactic types to semantic types can be defined. Using the types introduced in the previous chapter, let the correspondence be defined as:

$$\begin{aligned} \phi(S) &= s \rightarrow (s \rightarrow t) \\ \phi(E) &= e \\ \phi(b\backslash a) = \phi(a/b) &= \phi(b) \rightarrow \phi(a) \end{aligned}$$

This produces a type correspondence identical to the one used by Muskens [Mus91]. Furthermore, the definitions of $\phi(b\backslash a)$ and $\phi(a/b)$ reflect the intuition expressed in remark 3.1.

Let $G = \{(a, b) \in \Gamma \times \Lambda | \phi(a) = \tau(b)\}$. Then, Δ is **proper** iff the image of Δ is a subset of G. As a consequence of the Curry-Howard isomorphism [MR12], the following holds:

Lemma 3.1. *If Δ is proper and s is a derivable sentence, then $[\![s]\!]$ is a well-typed λ-term of type $\phi(\Psi(s))$ for all sentences s.*

The consequences are simple, yet powerful. Consider again the example *a man sleeps*. The calculation of $[\![a\ man\ sleeps]\!]$ can be illustrated through the derivation tree:

$$\dfrac{\dfrac{\dfrac{a}{[\![a]\!]} \qquad \dfrac{man}{[\![man]\!]}}{[\![a]\!][\![man]\!]} \qquad \dfrac{sleeps}{[\![sleeps]\!]}}{([\![a]\!][\![man]\!])[\![sleeps]\!]}$$

Consulting lemma 3.1, the tree corresponds to a tree of types such that the type of each term is consistent with the typing rules introduced in definition 2.1. The tree has the following form:

$$\dfrac{\dfrac{\dfrac{a}{e \rightarrow \sigma) \rightarrow ((e \rightarrow \sigma) \rightarrow \sigma)} \qquad \dfrac{man}{e \rightarrow \sigma}}{(e \rightarrow \sigma) \rightarrow \sigma} \qquad \dfrac{sleeps}{e \rightarrow \sigma}}{\sigma}$$

Since the sentence was well-formed, $([\![a]\!][\![man]\!])[\![sleeps]\!]$ has the type $\phi(S) = s \rightarrow (s \rightarrow t) = \sigma$ - which, as shown in the previous chapter, corresponds to a program of dynamic logic.

Based on ϕ, the creation of a dictionary can begin. I use the following categories and types for the syntactic classes:

Syntactic class:	**Category:**	**Type:**
Common noun	N	$e \to \sigma$
Proper noun	NP	$(e \to \sigma) \to \sigma$
Intransitive Verb	VP	$e \to \sigma$
Monotransitive Verb	$VP\ /\ NP$	$((e \to \sigma) \to \sigma) \to (e \to \sigma)$
Conjunction	$S\ \backslash(S/S)$	$\sigma \to (\sigma \to \sigma)$
Pronoun	NP	$(e \to \sigma) \to \sigma$
Relative Pronoun	$(N\ \backslash N\ )\ /\ VP$	$(e \to \sigma) \to ((e \to \sigma) \to (e \to \sigma))$
Article	NP/N	$(e \to \sigma) \to ((e \to \sigma) \to \sigma)$
Adjective	N/N	$(e \to \sigma) \to (e \to \sigma)$

Through the application of ϕ to every category defined in the list, the reader may assert that a dictionary based on this mapping is indeed proper. The mapping differs from the one presented by Muskens [Mus91, p.14] on three accounts.

I have added the syntactic class of *adjective*, using the category N/N. Since the type of N is $e \to \sigma$, the type of adjectives must be $(e \to \sigma) \to (e \to \sigma)$.

Muskens defines conjunctions as prefix operators by using the category $(S/S)/S$. Furthermore, an infix *if* with category $S\ \backslash(S/S)$ is defined. As the next section will show, the definition of *if* as an infix operator leads to an inconsistency. I presume that a mistake has led the two categories to be swapped. As such, I have elected to use the category $S\ \backslash(S/S)$ for conjunctions, and eliminated the prefix *if*.

Similarly, *who* is defined by Muskens as a prefix operator using the category $(N\ /\ N\ )\ /\ VP$. With such an interpretation, the grammatical way of using the pronoun becomes *a who man sleeps*. I define as an infix operator through the category $(N\ \backslash N\ )\ /\ VP$. This results in a grammatical usage consistent with common english: *a man who sleeps*.

3.2 The Problem of Causality

In the dictionary used by Muskens, causality is represented by the conjunction *if*. The following translation is given:

$$[\![if]\!] \equiv \lambda pq\lambda ij(i = j \wedge \forall h(pih \rightarrow \exists k(qhk))$$

Assuming that *if* is indeed a prefix operator, this corresponds to an understanding of *if* as either $[\![x \ if \ y]\!] \equiv [\![x]\!] \rightarrow [\![y]\!]$ or $[\![x \ if \ y]\!] \equiv [\![y]\!] \rightarrow [\![x]\!]$, depending on the assigned category.

The first interpretation reflects the category $(S/S)\backslash S$ used by Muskens. The result, however, seems contrary to the intuitive understanding of the word *if* in common English.

While the second interpretation - reflecting the category $(S\backslash S)/S$ - is consistent with common English, it does present a problem. Suppose that an anaphor in the second clause of the conjunction has an antecedent in the first - consider, for example, in the following sentence:

$$''A_1 \ \textit{farmer beats a donkey}_2 \ \textit{if it}_2 \ \textit{sleeps}''$$

The sentence $''\textit{it}_2 \ \textit{sleeps}''$ translates to:

$$\lambda ij(i = j \wedge \textit{sleeps} \ v_2 i)$$

Applying this to the semantics for *if*, we get:

$$\lambda q\lambda ij(i = j \wedge \forall h((i = h \wedge \textit{sleeps} \ v_2 i) \rightarrow \exists k(qhk))$$

The sentence $''A_1 \ \textit{farmer beats a donkey}_2''$ translates to:

$$\lambda ij(i[1,2]j \wedge \textit{farmer} \ v_1 j \wedge \textit{donkey} \ v_2 j \wedge \textit{beats} \ v_2 j \ v_1 j)$$

Combining this result with the former gives the following term:

$$\lambda ij(i = j \wedge \forall h((i = h \wedge \mathit{sleeps}\ v_2 i) \rightarrow \exists k(h[1,2]k \wedge \mathit{farmer}\ v_1 k \wedge \mathit{donkey}\ v_2 k \wedge \mathit{beats}\ v_2 k\ v_1 k))$$

The referent created by a_2 *donkey* appears later in the sentence than the reference it_2 - the reference is a cataphor, not an anaphor. As the example shows, the semantics defined by Muskens do not support cataphoric referents. Three strategies for repairing this issue are immediately come to mind:

- Change the semantics to support cataphors.
- Change the category of *if* to reflect a prefix notation, resulting in a translation as $[\![\mathit{if}\ x\ y]\!] \equiv [\![x]\!] \rightarrow [\![y]\!]$. This corresponds to the *if... then*-statement so common in programming languages.
- Introduce a new word with the meaning $[\![x]\!] \rightarrow [\![y]\!]$.

I have opted for the latter, and introduced the word *so* in the dictionary. I have attempted to come as close as possible to the intuitive $[\![x\ \mathit{so}\ y]\!] \equiv [\![x]\!] \rightarrow [\![y]\!]$. With that in mind, I define:

$$[\![\mathit{so}]\!] \equiv \lambda pq\lambda ij(i = j \wedge \forall h(qih \rightarrow \exists k(phk))$$

As a demonstration, consider the sentence A_1 *farmer beats* a_2 *donkey so* it_2 *hates* him_1. The sentence it_2 *hates* him_1 translates to:

$$\lambda ij(\mathit{hates}\ v_1 j\ v_2 j \wedge i = j)$$

Applying this to the semantics for *so*, we get:

$$\begin{aligned} [\![\mathit{so}\ it_2\ \mathit{hates}\ him_1]\!] &\equiv \lambda q\lambda ij(i = j \wedge \forall h(qih \rightarrow \exists k(\mathit{hates}\ v_1 k\ v_2 k \wedge h = k)) \\ &\equiv \lambda q\lambda ij(i = j \wedge \forall h(qih \rightarrow (\mathit{hates}\ v_1 h\ v_2 h)) \end{aligned}$$

As already mentioned, *"A_1 farmer beats a donkey$_2$"* translates to:

$$\lambda ij(i[1,2]j \wedge \mathit{farmer}\ v_1 j \wedge \mathit{donkey}\ v_2 j \wedge \mathit{beats}\ v_2 j\ v_1 j)$$

Combined with the former result, we get the following translation for A_1 *farmer beats* a_2 *donkey so* it_2 *hates* him_1:

$$\lambda ij(i = j \wedge \forall h((i[1,2]h \wedge farmer\ v_1h \wedge donkey\ v_2h \wedge beats\ v_2h\ v_1h) \rightarrow (hates\ v_1h\ v_2h))$$

Consulting the Θ-function, we see that the semantics correctly correspond to a test. Furthermore, the result corresponds to the intuitive definition $[\![so]\!] = [\![x]\!] \rightarrow [\![y]\!]$.

A similar issue exists with the conjunction *or*, though it is not of great consequence. Due to the semantics, a referent created in one clause cannot be picked up in another. As such, the following indexing becomes ungrammatical:

*"A_1 *woman sleeps or* she_1 *walks."*

Rather, the sentence results in the following translation:

A_1 *woman sleeps or* she_2 *walks."*

In English, this is not much of an issue - mostly, the first indexing would be written as *"A woman sleeps or walks"*. Therefore, while not decidedly ungrammatical, the first indexing is at least unusual. Preferring the second indexing, therefore, is somewhat natural.

3.3 Encyclopaedic Semantics

As more and more words are added to the dictionary, it becomes clear that the meaning of individual words are interdependent. As such, it is possible to define new words through existing words - or simply through logical terms. Jackendoff refers to this strategy as **encyclopaediac semantics**. [Jac02]

One attempt to capture this notion is description logics, in which semantics networks are used to emulate semantics dependencies.[JM09] While description logics are beyond the scope of this paper, it is interesting to see how well such phenomena are represented in the semantics used here.

Most words w in the dictionary have the effect of applying a predicate $p(w)$ to one or several entities. As explained, the predicate takes the form of a constant, to which the variables representing the entities are applied.

Definition 3.2. *Let Δ be a dictionary and w a word for which Δ is undefined. Furthermore, assume that w can be represented by an n-ary predicate $p(w)$. Then, a* ***description*** *$D(w)$ of w is a λ-term of the type $e_1 \to \ldots \to e_n \to s$ using no predicates undefined in Δ.*

To apply the description of a word w, simply replace $p(w)$ by $D(w)$ in $[\![w]\!]$.

To illustrate the concept, I have added a word *is* representing the interpretation of being as a form of equality between entities. As a monotransitive verb, the semantic function of the word gives:

$$[\![is]\!] \equiv \lambda Q \lambda y (Q \lambda x \lambda ij ((is\ y)x \wedge i = j))$$

Through definition 3.2, we define $D(is) = \lambda xy(x = y)$. Now, the constant *is* can be replaced by *D(is)*:

$$[\![is]\!] \equiv \lambda Q \lambda y (Q \lambda x \lambda ij (x = y \wedge i = j))$$

As an example, consider the sentence bob_1 *is a* $_2$ *farmer*. The expression *a* $_2$ *farmer* translates as:

$$\lambda P_2 \lambda ij \exists k (i[v_2]k \wedge \mathit{farmer}\ (v_2 k) \wedge P_2(v_2 k)kj)$$

Applying this to $[\![is]\!]$, we get:

$$\lambda y (\lambda ij (i[v_2]j \wedge \mathit{farmer}\ (v_2 j) \wedge (v_2 j) = y)$$

Applying this to the translation of Bob yields:

$$\lambda ij (v_1 i = bob \wedge i[v_2]j \wedge \mathit{farmer}\ (v_2 j) \wedge (v_2 j) = (v_1 i))$$

We see that the translation correctly applies the equality to the entities $v_1 i$ and $v_2 j$.

3.4 Defining the Semantic Function

Finally, the full dictionary can be defined. Using the types defined in section 3.1, the semantics of every word are defined below. Let x and y be variables of type e and let i through l be variables of type s. Furthermore, let p and q be variables of type σ, P_1 and P_2 be variables of type $e \to \sigma$ and let Q be a variable of type $(e \to \sigma) \to \sigma$. Then:

Word w**:**	**Corresponding Semantics** $[w]$**:**
Common noun cn	$\lambda x \lambda ij(cn\ x \wedge i = j)$
Proper noun pn_α	$\lambda P \lambda ij(v_\alpha i = pn\ \wedge P(v_\alpha i)ij)$
Pronoun p_α	$\lambda P \lambda ij(P(v_\alpha i)ij)$
Adjective a	$\lambda P \lambda x \lambda ij(Pxij \wedge a\ x)$
Intransitive verb v	$\lambda x \lambda ij(v\ x \wedge i = j)$
Monotransitive verb v	$\lambda Q \lambda y(Q \lambda x \lambda ij((v\ x)\ y \wedge i = j))$
is (monotransitive verb)	$\lambda Q \lambda y(Q \lambda x \lambda ij(x = y \wedge i = j))$
who (relative pronoun)	$\lambda P_1 P_2 \lambda x \lambda ij \exists h(P_2 xih \wedge P_1 xhj)$
and/. (conjunction)	$\lambda pq \lambda ij \exists h(pih \wedge qhj)$
or (conjunction)	$\lambda pq \lambda ij(i = j \wedge (\exists h(pih) \vee \exists k(qik)))$
so (conjunction)	$\lambda pq \lambda ij(i = j \wedge \forall h(pih \to \exists k(qhk)))$
a_α (article)	$\lambda P_1 P_2 \lambda ij \exists kh(i[v_\alpha]k \wedge P_1(v_\alpha k)kh \wedge P_2(v_\alpha k)hj)$
the_α (article)	$\lambda P_1 P_2 \lambda ij \exists k(P_1(v_\alpha k)ik \wedge P_2(v_\alpha k)kj)$
$every_\alpha$ (article)	$\lambda P_1 P_2 \lambda ij(i = j \wedge \forall kl((i[v_\alpha]k \wedge P_1(v_\alpha k)kl) \to \exists l(P_2(v_\alpha k)lh)))$
no_α (article)	$\lambda P_1 P_2 \lambda ij(i = j \wedge \neg \exists khl(i[v_\alpha]k \wedge P_1(v_\alpha k)kh \wedge P_2(v_\alpha k)hl))$

Finally, the semantic function is combined with the categories of section 3.1 to create the full dictionary Δ.

Chapter 4

Reduction and Simplification

Presuming that the semantic function correctly reflects the logical synthesis discussed in section 2.3, the semantics $[\![S]\!]$ of a sentence S should correspond to a program.

As the type of $[S]$ is $s \rightarrow (s \rightarrow t)$, beta-reduction can be used to reduce $[\![S]\!]$ to a term of the form λijR, where R is a term of PL_s. In order to further process this expression, three refinements take place as described in this chapter.

Section 4.1 and section 4.3 are adaptations of the strategy employed by Muskens [Mus91], while section 4.2 describes an additional step.

4.1 From Meaning to Truth

As a result of lemma 3.1, the translation process will result in a term of the type $s \rightarrow (s \rightarrow t)$. This corresponding to a relation between two states - in dynamic logic, a program. What we are interested in, however, is not the program itself - rather, we are interested in the set of states that could serve as initial state for the program. A further step is needed:

Definition 4.1. *The* ***content*** *of a sentence s is defined as* $\lambda i \exists j \; [\![s]\!] ij$.

Intuitively, definition 4.1 captures the aforementioned set. As Muskens points out, this definition corresponds to the step from discourse structure to truth taken in discourse representation theory [Mus91]. As such, it comes as no surprise when the same step is necessary here.

4.2 Anaphora as Generators of Entities

In corpora, anaphora often act as generators of referents. Mostly, this phenomenon occurs when no previous reference exists, e.g. when referents are introduced by name (*"Alice loves bob."*) or a text starts *in medias res* (*"The elevator continued its horribly slow ascent.", "He had known the woman all his life."*).

The content of any well-formed sentence will be a term on the form $\lambda i \exists j \; R$. In the cases discussed, the expression R will contain references to the values of stores in state i - the first state of execution. Muskens' solution is to require a deictic antecedent for every such store. [Mus91] As the given examples show, finding one such is impossible in many cases. Therefore, we desire a definition of content independent of the initial state.

Assume that $v_1, ..., v_n$ is the sequence of stores referenced in the initial state i in R. Let α be any state, and let β be a state such that $\alpha[v_1, ..., v_n]\beta$. Then the existence of β is guaranteed by the update axiom. As such, the ontological assumption that every deictic reference has some referent can be formulated through the following definition:

Definition 4.2. *The* ***non-deictic content*** *of a sentence s is defined as* $\lambda \alpha \exists \beta k \; (\alpha[v_1, ...v_n]\beta \wedge [\![s]\!]\beta k)$.

Crucially, this definition is equivalent to definition 4.1 apart from the aforementioned ontological assumption of non-deixis.

4.3 From States to Entities

By necessity, the non-deictic content of any well-formed sentence will be reducible to the form $\lambda i\ R$, where R is a term of PL_s. Expressive as PL_s is, states and stores prove a needless complication after translation is complete. As such, a translation into PL_e is desired. To remedy the situation, Muskens offers the **Unselective Binding Lemma** [Mus91]

Lemma 4.3. *Let $u_1, ...u_n$ be store names, let $x_1, ..., x_n$ be distinct variables, and let φ be a formula that does not contain j. Then for all states i:*

1. $\exists j(i[u_1, ..u_n]j \wedge \varphi[x_1/u_1 j, ..., x_n/u_n j])$ *is equivalent to* $\exists x_1...x_n\varphi$
2. $\forall j(i[u_1, ..u_n]j \rightarrow \varphi[x_1/u_1 j, ..., x_n/u_n j])$ *is equivalent to* $\forall x_1...x_n\varphi$

Muskens omits the proof of the lemma, as it is rather simple. For the sake of completeness, I include it here:

Proof. To show that 1 holds, assume that φ is true in some state i. Then, $d_1, ..., d_n$ exist which satisfy ϕ as substitutions for $x_1, ..., x_n$. By the Update Axiom introduced by Muskens [Mus91], there exists a state j such that $i[u_1, ...u_n]j$ and $u_k j = d_k$ for all k between 1 and n. Hence, $\exists j(i[u_1, ..u_n]j \wedge \varphi[x_1/u_1 j, ..., x_n/u_n j])$ holds in i.

Now assume that $\exists j(i[u_1, ..u_n]j \wedge \varphi[x_1/u_1 j, ..., x_n/u_n j])$ is true in some state i. Then by necessity the entities $u_1 j, ..., u_n j$ satisfy φ. Hence, $\exists x_1...x_n\varphi$ holds. □

Proof. That 2 follows from 1 is easily shown:

$$\begin{aligned}
\forall j(i[u_1, ..u_n]j \rightarrow \varphi[x_1/u_1 j, ..., x_n/u_n j]) &\equiv \neg\exists j\neg(i[u_1, ..u_n]j \rightarrow \varphi[x_1/u_1 j, ..., x_n/u_n j]) \\
&\equiv \neg\exists j(i[u_1, ..u_n]j \wedge \neg\varphi[x_1/u_1 j, ..., x_n/u_n j]) \\
&\equiv \neg\exists x_1...x_n\neg\varphi \\
&\equiv \forall x_1...x_n\varphi
\end{aligned}$$

□

Through repeated application of lemma 4.3, $\lambda i\ R$ can be translated to an expression of the form $\lambda i\ Q$, where Q is a term of PL_e. That this is the case follows from the form of the dictionary. State-changing terms of the form $s_1[v_n]s_2$ are only introduced through $[\![a]\!]$, $[\![no]\!]$ and $[\![every]\!]$. In all cases, s_2 is bound by a

quantifier figuring in the same term - lemma 4.3 can be applied directly if the sub-terms are known. Any reference to a state must be either a reference to a quantified state or a reference to the initial state. In the first case, lemma 4.3 ensures that the store reference is replaced with an entity. In the second case, the use of definition 4.2 guarantees that no such reference can exist.

CHAPTER 5

Filtering of Indexings

Determining whether an indexing is suitable can be done by a combination of checks. I include three such filters - syntactic, gender-based and nominal. In the following sections I define three conditions, that together ensure a level of consistency in the accepted indexings: Definition 5.2, definition 5.3, and definition 5.4.

Provided that these conditions are upheld, a large fragment of the language defined by Δ can be indexed unambiguously.

5.1 Syntactic Filtering

As evidenced by example 1 and 2, co-indexing may directly determine the well-formedness of sentences. Furthermore, although the semantic value of subject-, object- and reflexive object-pronouns may be identical, their syntactic positions are restricted as examples 3 and 4 show.

1. *"$John_1$ hit him_1."*
2. *"$John_1$ hit $himself_2$."*

3. *"He_1 hit she_2."

4. *"Him_1 hit her_2."

This phenomena is explained at length by the Government and Binding theory presented by Chomsky. [Cho93] To fully describe the dictionary used in this paper, a simplified theory suffices.

To define the relevant aspects of the theory, the concept of **c-command** is introduced. A node A in a derivation tree c-commands another node B if and only if:

- A is not an ancestor of B.
- B is not an ancestor of A.
- The first branching node that is an ancestor of A is an ancestor of B.

In the language of categorial grammars:

Definition 5.1. *Let A and B refer to nodes in the derivation tree of a sentence s. Let N_1, and N_2 be sequences of categories and let X and Y be categories. Then A c-commands B iff $B \in N_2$ and $\Psi(s)$ contains either of the following derivative steps:*

$$(1): \frac{N_1 \vdash A \qquad N_2 \vdash Y}{N_1, N_2 \vdash X} \qquad (2): \frac{N_2 \vdash Y \qquad N_1 \vdash A}{N_1, N_2 \vdash X}$$

B is **governed** by A if and only if A c-commands B and A is of the category *NP*. B is **bound** by A if and only if A governs B and A and B corefer.

Definition 5.2. *A sentence is defined to be* ***syntax-consistent*** *if and only if the following constraints hold:[All95]*

1. *Subject pronouns cannot be governed.*
2. *Object pronouns must be governed.*
3. *Reflexive pronouns must be bound. Any other anaphora cannot be bound.*

With the constraints in mind, consider examples 1 through 4. In example 1, *him* is bound by *John*, violating rule 3. Similarly, in example 2, rule 3 is violated as *himself* is not bound. Example 3 violates rule 1 as *she* is governed. Finally, example 4 violates rule 2 as *him* is not governed.

5.2 Gender-based Filtering

In English as in many other languages, pronouns are filtered by gender. Indeed, the proper interpretation of a text often hinges on gendered information. Consider the following sentences:

- *"A_1 farmer owns a_2 donkey. He_1 beats it_2"*
- *"A_1 farmer owns a_2 donkey. She_1 beats it_2"*
- **"A_1 farmer owns a_2 donkey. It_1 beats it_2"*

Intuitively, indexings one and two are grammatical, while indexing three is not. The explanation is that a variable is referenced by two different genders.

As such, there is a clear parallel to type constraints in programming languages - integer-variables should not be referenced as string-variables, and so on.

In accordance with this line of reasoning, Ranta suggests an intricate system of subtypes. [Ran94] To reflect the genders of English, three subtypes of entities are introduced: e_{male}, e_{female} and e_{neuter}, corresponding to "he", "she" and "it".

To limit the genders of a word, the semantics may restrict the type. For example, the type of a woman must be e_{female}:

$$[\![woman]\!] \equiv \lambda x : e_{female} \lambda ij : s\ (woman\ x \wedge i = j)$$

While all pronouns are gendered, the same is not true of all nouns. For example, a woman can only refer to female entities, while a farmer can refer to entities of either gender - though not neutral entities. Similarly, there are names which can be used of both men and women (Alex, Quinn, Elliot, Riley). In such cases, it may be preferable to exclude one gender by the use of a negation, or provide no gender-information at all. To enable this possibility, one can draw upon the interpretation of types as sets. As such, the type of a farmer should be $e \backslash e_{neuter}$.

In practise, it is simpler to disassociate the genders from other type information. To this purpose, the set of possible genders of an entity e in a term t is expressed as a function $G(e, t)$. If l is a leaf where e is constrained to the set of genders L, $G(e, l) = L$. If l is a leaf where e is not constrained,

$G(e, l) = \{male, female, neuter\}$. For a composite expression t_{comp} with children t_1 and t_2, $G(e, t_{comp}) = G(e, t_1) \cap G(e, t_2)$. For a non-composite expression t_{parent} with a child t_{child}, $G(e, t_{parent}) = G(e, t_{child})$. With this in mind, define:

Definition 5.3. *An indexing on a term T is **gender-consistent** if and only if $G(e, T) \neq \emptyset$ for all proper nouns and for all entities quantified in T.*

Requiring every indexing to be gender-consistent is equivalent to the extension of the type-system with subtypes. The proof is trivial, especially with Curry's definition of typed terms.

5.3 Nominal Filtering

Proper nouns present another criteria upon which to filter indexings. It is clear that the following indexing cannot be correct:

**"Alice$_1$ sleeps and Clara$_1$ walks."*

Rather, the proper indexing should be:

"Alice$_1$ sleeps and Clara$_2$ walks."

The reason is that Alice and Clara due to being differently named must refer to different entities. I define this form of consistency as follows:

Definition 5.4. *An indexing on a sentence s is **name-consistent** if and only if the deictic content of s does not contain an expression of the form $n_1 = n_2$, where $n_1, n_2 \in Names$.*

Due to the demand that all entity generators use unique indexing numbers, an entity will only be referred to by two different proper nouns if two otherwise separate entities were equalised through use of *is*. Therefore, a sentence s contains an contains an entity referred to by two different proper nouns iff $[\![s]\!]$ contains an expression of the form $n_1 = n_2$ for two names n_1 and n_2.

It is important to note that corpora sometimes contain examples where an entity *can* have multiple names - for example a last name and a first name. As such, a more thorough investigation of the naming of entities would be required for full dictionaries. In this case, the creation of a database for entities would be sensible.

Chapter 6

Implementation

In this section I shall explain the workings of the *proof of concept* program accompanying this thesis. As a consequence of my initial intention to use the Isabelle proof assistant in the implementation of a logical filter, I have written the program in the Poly/ML language (version 5.5), a full implementation of Standard ML [Mat12].

The program is structured around a file *Interface.ML*, which imports and connects every other component. The interface offers four functions:

- **translate**, which given a sentence s returns a string-representation of the non-deictic content of s through generation and filtering of indexings.
- **translate_indexed**, which given a sentence s and an indexing i returns a string-representation of the non-deictic content of s using i as indexing.
- **translate_plain**, which given a sentence s and an indexing i returns a string-representation of the content of s using i as indexing - without any form of simplification.
- **meaning_of**, which given a word w returns $[\![w]\!]$.

Furthermore, a test suite has been implemented, offering a function **full_test** to perform a large set of assertions on the output of the program.

6.1 Data Structures

Categories are represented using a datatype, with / and \ defined as infix operators:

```
datatype Category =
    E
    | S
    | \ of Category*Category
    | / of Category*Category
infixr \;
infixr /;
```

The terms of Λ are represented by the datatype shown below. Note that in addition to those discussed in section 2.1, a constant *Tr* has been added, representing a tautology. To simplify the terms, implication has not been implemented - rather, $p \rightarrow q$ should be represented as $\neg p \vee q$.

```
datatype Term = Var of string
    | Abs of string*Term
    | App of Term*Term
    | Exists of string*Term
    | Forall of string*Term
    | Store of int*string
    | Conj of Term*Term
    | Eq of Term*Term
    | Differs of int list*string*string
    | Not of Term
    | Disj of Term*Term
    | Tr
```

The dictionary is represented by a tokenization function. As such, tokens corresponding to triples of $w, \Delta(w)$ for every word w are necessary:

```
datatype Token = T of string*Category*Term
```

Finally, derivation trees are represented through a simple binary tree structure:

```
datatype ParseTree =  Fail
                    | Leaf of Category*Token
                    | Node of Category*ParseTree*ParseTree;
```

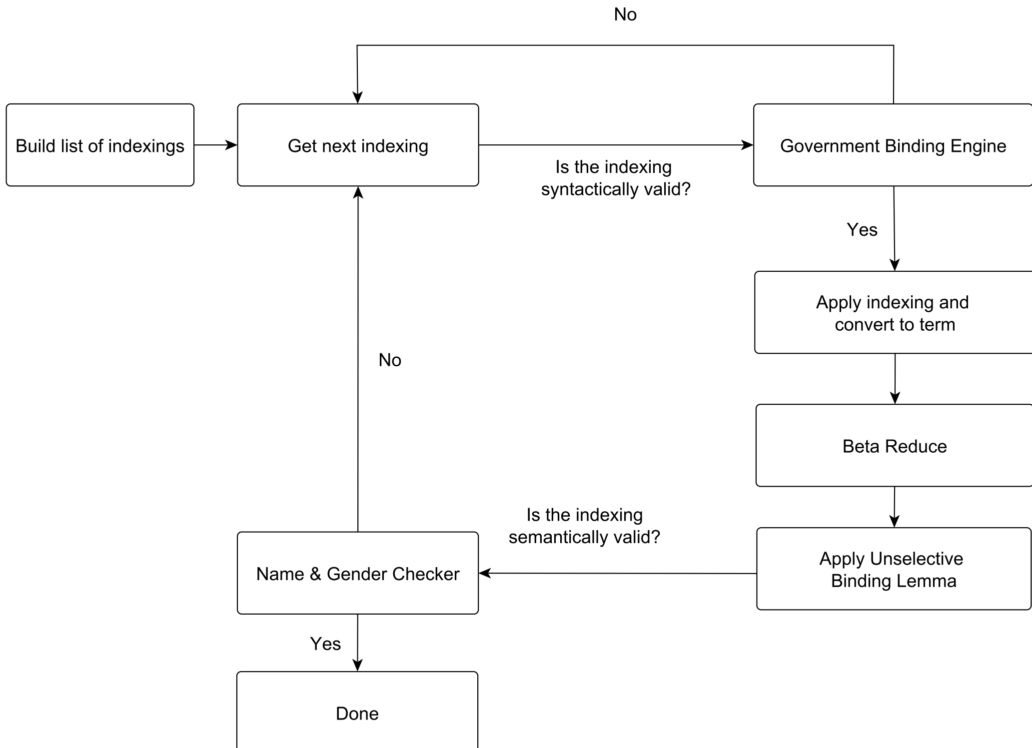
Note the special node *Fail*. This represents a branch of a tree being ungrammatical.

6.2 Framework

When a sentence is input to the program, it is initially converted to a list of tokens by the lexer. Afterwards, a list of potential indexings are generated. Then, the list is traversed, building translations in an attempt to validate one.

During preprocessing, a function *generate_possibilities* builds a list of all potentially viable indexings based on the total amount of references. This is carried out by an algorithm inspired by the digital clock; the next indexing is generated by finding the first reference to differ from the total amount of references and incrementing that, resetting every earlier referent to 1. As described in the introduction, generating references are locked to a constant index.

Once the entire list of indexings has been built, an algorithm roughly corresponding to the following flowchart is used:



6.3 Tokenization

The Lexer

The lexer reads the input character by character. When a space or period is met, the encountered letters are converted to a token using the *tokenize*-function.

One notable exception is the period. If a period is the last word of the text, it is ignored. If, however, more words follow upon a period, it is read as the word *"and"*. This is possible as the translation for *"."* and *"and"* are identical.

The Dictionary

The dictionary consists of a function $Tokenize : string \to Token$. Most words can be described using standardized or semi-standardized syntactic and semantic functions, such as in the following example:

```
| tokenize "sleeps" = T("sleeps", verb_intransitive,
    verb_intransitive_term "sleeps")
```

Note that all information pertaining to store numbers is left to be added later.

Some words - notably those acting as articles or conjunctions - require customized translations:

```
| tokenize "the" = T("the",article,
   let
       val v1 = get_new_var()
       val v2 = get_new_var()
       val v3 = get_new_var()
       val v4 = get_new_var()
       val v5 = get_new_var()
   in
       Abs(v1, Abs(v2,Abs(v3,Abs(v4, Exists(v5,
            Conj(App(App(App(Var(v1), Store(0, v5)), Var(v3)), Var(v5)),
            App(App(App(Var(v2), Store(0, v5)), Var(v5)), Var(v4)))))))
   end
)
```

If a word occurs which is not present in the dictionary, an *UnknownWord*-exception is raised.

6.4 Parsing

A *derivation_tree* is built based on the list of tokens. The tree is constructed using a simple backtracking-algorithm. First, each token in the list is encapsulated in a leaf. Afterwards, the *build_tree* function is called:

```
fun build_tree [] = Fail
| build_tree [Elem] = Elem
| build_tree l =
    if (has_derivation l)
    then try_all (find_derivation_list [] l)
    else Fail
```

A function *find_derivation_list* constructs a list of all possible derivations from a given sequence of trees. Then, the function *try_all* attempts to recursively derive a complete tree from each possibility:

```
fun try_all [] = Fail
| try_all (x::xs) =
    let
        val res = build_tree x
    in
        if res = Fail then try_all xs else res
    end
```

If at any point a derivation is impossible, a node of the type *Fail* is returned, signifying a grammatical error.

Single derivations are handled through a combination of functions: *deriv_cat_possible*, *deriv_node_possible*, *deriv_cat_result*, and *deriv_node_result*. The first two return boolean values representing whether a derivation rule can be applied to two categories. The other represent the result.

As pointed out in remark 3.1, it is necessary to alter the ordering of the nodes when a \-derivation is used. In the implementation, this is represented through a boolean value:

```
fun deriv_cat_result (a/b) (c\d) =
    if (b = (c\d)) then (a, false)
    else if (c = (a/b)) then (d, true)
    ...
```

The boolean is used to construct a tree of the correct form:

```
fun deriv_node_result (Node(c1,t1,t2)) (Node(c2,t3,t4)) =
    let
        val (res, switch) = deriv_cat_result c1 c2
    in
        if switch then
            Node(res, (Node(c2,t3,t4)), (Node(c1,t1,t2)))
        else
            Node(res, (Node(c1,t1,t2)), (Node(c2,t3,t4)))
    end
...
```

6.5 Beta-Reduction

Once the parse tree is constructed, a direct translation to lambda calculus is possible. All leaves are replaced by a lambda term determined by the referenced token:

```
fun to_term (Leaf(_, T(_,_,t))) = t
```

All inner nodes are recursively replaced by an application of the left child to the right child:

```
| to_term (Node(_, t1, t2)) = App((to_term t1), (to_term t2))
```

Two functions *replace* and *beta_reduce* are implemented, representing respectively α-conversion and β-reduction. *Replace* is a simple recursive traversal:

```
fun replace (Var(x)) s t2 = if x = s then t2 else Var(x)
| replace (App(te1, te2)) s t2 = App( (replace te1 s t2), (replace te2 s
    t2))
...
```

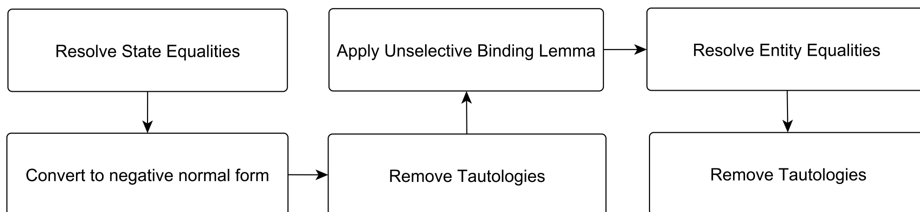
The implementation of beta-reduction is similar to the theoretical definition:

```
beta_reduce (App(Abs(s, t1), t2)) = beta_reduce (replace t1 s t2)
| beta_reduce (App(t1, t2)) =
    let
        val res = beta_reduce t1
    in
        if res = t1
        then App(t1,t2)
        else beta_reduce (App(res, t2))
    end
...
```

Note that when an application where the β-reduction rule cannot be applied is encountered, forward recursion is used to ensure that the inner term is reduced as widely as possible.

6.6 Simplification & Unselective Binding Lemma

The simplification process consists of five different functions, some of which are applied multiple times. The full process can roughly be described by the following flowchart:



In the following sections, each individual function shall be described. Note that the application of the Unselective Binding Lemma is a two-part process - first, the term is rewritten to use the shorthand notation $i[v1_1, ..., v_n]j$ as devised by Muskens [Mus91]. Then, the lemma is applied.

Resolution of Equalities

Any formula of the form $\exists y\, t$ is replaced by $t[x/y]$ if $x = y$ in t. Finding equalities within a term is done on a set basis, ensuring that equalities in disjunctions and implications are located:

```
fun find_equality x (Eq(Var(s1), Var(s2))) =
    if ((x = s1) andalso (s1 <> s2)) then singleton s2 else
        if ((x = s2) andalso (s1 <> s2)) then singleton s1 else empty_set
| find_equality x (Conj(t1, t2)) = set_union (find_equality x t1)
    (find_equality x t2)
| find_equality x (Exists(_, t)) = find_equality x t
...
```

Importantly, the function employs forward recursion. This ensures that the quantifiers at the highest possible level are maintained.

As a consequence of the structure of the semantics, all quantified variables will be of the type s before lemma 4.3 is applied. Similarly, they will be of the type e afterwards. Therefore, this function can be reused to unify entities who have been declared equal through *is*.

The second application of this process should happen before nominal filtering, but after application of lemma 4.3. Otherwise, expressions like *"*$alice_1$ *is* a_2 *farmer.* The_2 *farmer is* bob_3*.* could result in wrongful indexings.

As equality resolutions leaves a trail of tautological expressions throughout the term, the tautology-removal function is applied once after each time equality resolution is used.

Convertion to Negative Normal Form

The expression is put on negative normal form through application of De Morgan's laws and the definition of the quantifiers:

```
...
(*Negated quantors:*)
| push_negations (Not(Exists(s,x))) = Forall(s, push_negations (Not(x)))
| push_negations (Not(Forall(s,x))) = Exists(s, push_negations (Not(x)))

(*De Morgan's Laws:*)
| push_negations (Not((Disj(t1,t2)))) = Conj(push_negations
    (Not(t1)),push_negations (Not(t2)))
```

```
| push_negations (Not((Conj(t1,t2)))) = Disj(push_negations
    (Not(t1)),push_negations (Not(t2)))
...
```

While not strictly necessary, the use of negative normal form make the rest of the simplification process easier, as negations are out of the picture. Furthermore, it has the added benefit of applying the following equivalence to implications:

$$(i[...]j \wedge x) \rightarrow y \equiv i[...]j \rightarrow (x \rightarrow y)$$

This translation simplifies the extraction process described in the next section.

Extraction of State Information

Using the following definition, expressions of the form $i[x]j$ are shortened as described by Muskens [Mus91].

Definition 6.1. *A successor to an existentially quantified variable i of type s in a term t is defined as an existentially quantified variable j such that $t \equiv i[x]j \wedge y$ for some x and y.*

A successor to a universally quantified variable i of type s in a term t is defined as a universally quantified variable j such that $t \equiv i[x]j \rightarrow y$ for some x and y.

The list of successors to a variable k is created by finding a successor to l to k, then finding a successor to l, and so on until no successor exists:

```
fun existential_successor_list term var used_stores =
    let
        val (store, state) = existential_successor term var
    in
        if (store <> 0) andalso (exist_quant state term) andalso
            (not(contains used_stores store))
        then (store, state)::(existential_successor_list term state
            (store::used_stores))
        else []
    end
```

Based on this list, an expression $s_1[x_1, x_2, ...x_n]s_2$ is created. A successor s_n is discounted for a state i if $s_{n-1}[x_n]s_n$ and $i[x_1, ..., x_n, ...]s_{n-1}$. This ensures that no store can be changed twice by the same expression.

Once a list $s_1[x_1, x_2, ...x_n]s_2$ has been created for some pair of states, all store references using an intermediate state are pushed forward to s_2 or backwards to s_1, depending on whether the store referenced has been updated when it is encountered. This is done by mapping store references to states such that if a store p changes value in an intermediate state q, p is mapped to q. If a reference to p is found in a predecessor-state to q, the reference is changed to the initial state s_1 otherwise, it is changed to the final state s_2:

```
fun push_stores (Store(i, s)) mapping top_var =
    if (contains_state s mapping) then
        if (earlier i s mapping) then (Store(i, last_state mapping))
        else (Store(i, top_var))
    else (Store(i, s))
...
```

Finally, the term in question is updated to reflect the changes. Quantifiers pertaining to intermediate states and old difference-expressions are removed, and a new difference-expression is added. If the successors are existentially quantified, the resulting term becomes $\exists s_2(s_1[...]s_2 \wedge t)$:

```
Exists(last, state_update (Conj((diff), new_term)) last)
```

Otherwise, if the successors are universally quantified, the resulting term becomes $\forall s_2(s_1[...]s_2 \rightarrow t)$:

```
(Forall(last, (Disj(Not(diff), state_update new_term last)))
```

Removal of Tautologies

The removal of tautologies is straightforward. Equalities are replaced with *Tr* if they are of the form $x = x$:

```
fun taut_replace (Eq(t1, t2)) = if t1 = t2 then Tr else (Eq(t1, t2))
```

Conjunction, disjunction and negation are updated to reflect normal logical rules, such that:

- $\neg T \wedge x \equiv \neg T$
- $\neg T \vee x \equiv x$
- $T \wedge x \equiv x$

- $T \vee x \equiv T$

This is reflected in the code as follows:

```
| taut_replace (Conj(t1, t2)) =
    let
        val left = taut_replace t1
        val right = taut_replace t2
    in
        if left = Tr then right
        else if right = Tr then left
            else if (left = Not(Tr)) orelse (right = Not(Tr)) then Not(Tr)
                else (Conj(left, right))
    end
...
```

Application of Unselective Binding Lemma

The Unselective Binding Lemma is used to replace quantifications over states with quantifications over variables.

As state information was extracted to a high a level as possible, the application of the binding lemma is simple. A function *gather_ vars* collects a list of variables if the lemma should be applied:

```
fun gather_vars (Differs(i, s1, s2)) s4 =
    if (s2 = s4) then generate_var_list i
    else []
...
```

As shown, the function *generate_ var_ list* is called. This generates a new variable for each store referenced:

```
fun generate_var_list [] = []
| generate_var_list (x::xs) = (x, get_new_var())::(generate_var_list xs)
```

The new variables are used as described in Anaphora and the Logic of Change to create an expression of the form $\exists x_1 ... x_n \phi$ or $\forall x_1 ... x_n \phi$.

6.7 Validation of Indexings

Government and Binding Checker

A tree-walking algorithm is used to determine whether the chosen indexing is consistent with Government and Binding theory. The traversal is performed mainly by the *government_ binding*-function:

```
fun government_binding (Leaf(_,T(_,_,_))) = true
| government_binding (Node(c,t1, t2)) =
    if (conj t1) orelse (conj t2)
    then (government_binding t1) andalso (government_binding t2)
    else
        if conj (Node(c,t1, t2))
        then government_binding t2
        else (free t1) andalso (governed t2 (get_governor t1))
...
```

The function distinguishes between two kinds of inner nodes: Those which correspond to a conjunction and those which do not. Due to the nature of the implemented grammar, it holds for all derivable sentences that the root node of any composite derivation tree must be either a conjunction or a verb. Furthermore, the children of a conjunction-node are either conjunction-nodes or verb-nodes. Therefore, this simplification is applicable.

If a verb node is encountered, it must hold that the left child governs of the right child. A function *get_ governor* is used to extract the governing entity. Then, two functions *free* and *governed* are used to ensure that the rules set out in definition 5.2 hold.

Nominal Filtering

Ensuring that no entity is referred to by two different names is a simple process - the expression is traversed, searching for equality-expressions. If such an expression is found, the program requests a new indexing if the children contain references to two different names.

Gender-based Filtering

The semantics of every noun, pronoun or proper noun has been extended with an additional term describing gender. As an example, consider the noun *"farmer"*. Without gender information, the semantics for a noun are implemented as:

$$\lambda x \lambda ij(i = j \wedge noun\ x)$$

The term is expanded with an expression *gender*, to which the subject is applied:

$$\lambda x \lambda ij(i = j \wedge gender\ x \wedge noun\ x)$$

As an example, assume that the gender of *"farmer"* may be either male or female, but not neuter. The corresponding term is $\lambda e \neg(neuter\ e)$. Replacing the *gender*-variable in the semantics for noun results in the following term:

$$\lambda x \lambda ij(i = j \wedge \neg(neuter\ x) \wedge noun\ x)$$

Pronouns and proper nouns are implemented in similar fashion.

The function G defined in definition 5.3 is implemented through two functions *gender_check_inner* and *gender_check_traverse*. For predicates, G is defined as:

```
gender_check_inner s (App(Var(x), Var(s2))) =
    if (s = s2) andalso (((x = "male") orelse (x = "female")) orelse (x
        = "neuter"))
    then singleton x
    else ["male", "female", "neuter"]
| gender_check_inner s (App(Not(Var(x)), Var(s2))) =
    if (s = s2) andalso (((x = "male") orelse (x = "female")) orelse (x
        = "neuter"))
    then set_minus ["male", "female", "neuter"] (singleton x)
    else ["male", "female", "neuter"]
...
```

For composite expressions, the implementation directly copies the rules set forth in section 5.2:

```
...
| gender_check_inner s (Conj(t1, t2)) = set_intersection
    (gender_check_inner s t1) (gender_check_inner s t2)
...
```

The outer function - *gender_ check_ traverse* - repeatedly uses the inner to manage the gender of every quantified entity and every proper noun:

```
fun gender_check_traverse (Exists(s, t)) =
    let
        val res = gender_check_inner s t
    in
        if res = empty_set then false
        else gender_check_traverse t
    end
```

Chapter 7

Examples

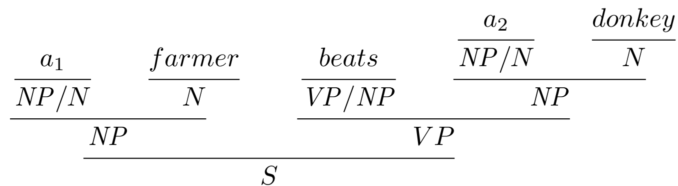
In this chapter, I shall present various examples illustrating the implementation. In the first section, a detailed example will be shown by hand. In the second section, several examples of interest will be highlighted, although only the final translation will be shown.

7.1 A Detailed Example

Consider the following sentence:

"A_1 *farmer beats* a_2 *donkey. the*$_a$ *poor animal hates him*$_b$"

The first part of the text - A_1 *farmer beats* a_2 *donkey* - corresponds to the following parse tree:



Applying the translation of *farmer* to the translation of a_1, we get:

$$\begin{aligned}
[\![a_1 \ farmer]\!] &= (\lambda P_1 P_2 \lambda ij \exists kh (i[v_1]k \wedge P_1(v_1k)kh \wedge P_2(v_1k)hj)) \\
&\quad \lambda x \lambda ij \ (farmer \ x \neg \wedge neuter \ x \wedge i = j) \\
&= \lambda P_2 \lambda ij \exists kh (i[v_1]k \wedge \lambda x \lambda ij \ (farmer \ x \neg \wedge neuter \ x \wedge i = j)(v_1k)kh \\
&\quad \wedge P_2(v_1k)hj) \\
&= \lambda P_2 \lambda ij \exists kh (i[v_1]k \wedge (farmer \ (v_1k) \wedge \neg neuter \ (v_1k) \wedge k = h) \\
&\quad \wedge P_2(v_1k)hj) \\
&= \lambda P_2 \lambda ij \exists k (i[v_1]k \wedge farmer \ (v_1k) \wedge \neg neuter \ (v_1k) \wedge P_2(v_1k)kj)
\end{aligned}$$

In similar fashion:

$$[\![a_2 \, donkey]\!] = \lambda P_2 \lambda ij \exists k (i[v_2]k \wedge donkey \ (v_2k) \wedge neuter \ (v_2k) \wedge P_2(v_2k)kj)$$

The meaning of *beat* a_a *donkey* is derived as:

$$\begin{aligned}
[\![beats \ a_2 \ donkey]\!] &= (\lambda Q \lambda y (Q \lambda x \lambda ij (beats \ xy \wedge i = j))) \\
&\quad \lambda P_2 \lambda ij \exists k (i[v_2]k \wedge donkey \ (v_2k) \wedge neuter \ (v_2k) \wedge P_2(v_2k)kj) \\
&= \lambda y (\lambda ij \exists k (i[v_2]k \wedge donkey \ (v_2k) \wedge neuter \ (v_2k) \wedge beats \ (v_2k)y \\
&\quad \wedge k = j)) \\
&= \lambda y (\lambda ij (i[v_2]j \wedge donkey \ (v_2j) \wedge neuter \ (v_2j) \wedge beats \ (v_2j)y)
\end{aligned}$$

Finally, applying $[\![beat \ a_a \ donkey]\!]$ to $[\![a_1 \ farmer]\!]$, the following is derived. Note that all state variables are pushed as far as possible forward, i.e. v_1k is replaced by v_1j since $k[v_2]j$.

$$\begin{aligned}
[\![a_1 \ farmer \ beats \ a_2 \ donkey]\!] &= \lambda ij \exists k (i[v_1]k \wedge farmer \ (v_1k) \wedge \neg neuter \ (v_1k) \\
&\quad \wedge (k[v_2]j \wedge donkey \ (v_2j) \wedge neuter \ (v_2j) \\
&\quad \wedge beats \ (v_2j)(v_1k)) \\
&= \lambda ij (i[v_1, v_2]j \wedge farmer \ (v_1j) \wedge \neg neuter \ (v_1j) \\
&\quad \wedge donkey \ (v_2j) \wedge neuter \ (v_2j) \\
&\quad \wedge beats \ (v_2j)(v_1j))
\end{aligned}$$

In completely analogous fashion, the sentence the_a *poor animal hates* him_b corresponds to:

$$\dfrac{\dfrac{\dfrac{the_a}{NP/N} \quad \dfrac{\dfrac{poor}{N/N}\ \dfrac{animal}{N}}{N}}{NP} \qquad \dfrac{\dfrac{hates}{VP/NP} \quad \dfrac{him_b}{NP}}{VP}}{S}$$

By the same translation procedure, $[\![the_a\ poor\ animal\ hates\ him_b]\!]$ is derived:

$$[\![the_a\ poor\ animal\ hates\ him]\!] = \lambda i \lambda j(poor(v_a i) \wedge animal \wedge neuter$$
$$(v_a i) \wedge hates(v_b i)(v_a i) \wedge male(v_b i) \wedge i = j)$$

Combining the two sentences using $[\![.]\!]$ gives the following translation:

$$\lambda ij(i[v_1, v_2]j \wedge farmer\ (v_1 j) \wedge \neg neuter\ (v_1 j) \wedge donkey\ (v_2 j) \wedge neuter\ (v_2 j)$$
$$\wedge\ beats\ (v_2 j)(v_1 j) \wedge poor(v_a j) \wedge animal(v_a j) \wedge neuter(v_a j)$$
$$\wedge\ hates(v_b j)(v_a j) \wedge male(v_b j))$$

The non-deictic content of the combined sentence is then:

$$\lambda i \exists j(i[v_1, v_2]j \wedge farmer\ (v_1 j) \wedge \neg neuter\ (v_1 j) \wedge donkey\ (v_2 j) \wedge neuter\ (v_2 j)$$
$$\wedge\ beats\ (v_2 j)(v_1 j) \wedge poor(v_a j) \wedge animal(v_a j) \wedge neuter(v_a j)$$
$$\wedge\ hates(v_b j)(v_a j) \wedge male(v_b j))$$

As $i[v_1, v_2]j$, a maximum of two stores are used in the interpretation of the combined sentence. Consider the variables a and b. If $a = b$, the second part of the sentence has *the poor animal* binding and governing *him* - breaking the rules of Government and Binding Theory. That gives two possible interpretations:

$$\lambda i \exists j(i[v_1, v_2]j \wedge farmer\ (v_1 j) \wedge \neg neuter\ (v_1 j) \wedge donkey\ (v_2 j) \wedge neuter\ (v_2 j)$$
$$\wedge\ beats\ (v_2 j)(v_1 j) \wedge poor(v_1 j) \wedge animal(v_1 j) \wedge neuter(v_1 j)$$
$$\wedge\ hates(v_2 j)(v_1 j) \wedge male(v_2 j))$$

Or:

$$\lambda i \exists j (i[v_1, v_2]j \wedge farmer\ (v_1 j) \wedge \neg neuter\ (v_1 j) \wedge donkey\ (v_2 j) \wedge neuter\ (v_2 j)$$
$$\wedge\ beats\ (v_2 j)(v_1 j) \wedge poor(v_2 j) \wedge animal(v_2 j) \wedge neuter(v_2 j)$$
$$\wedge\ hates(v_1 j)(v_2 j) \wedge male(v_1 j))$$

In the first of the two, $\neg neuter\ (v_1 j) \wedge neuter(v_1 j)$, giving rise to contradiction in the gender-based filtering. As such, only the second of the two is viable. Applying the *unselective binding lemma*, the following term is reached and finally returned, after gender-information has been removed:

$$\exists xy(farmer\ x \wedge donkey\ y \wedge beats\ yx \wedge poor\ y \wedge animal\ y \wedge hates\ xy)$$

7.2 Other Interesting Cases

In this section, the results of various interesting translations will be shown. Note that as the simple cases such as *"a man sleeps"* have mostly been examined by hand throughout the paper, they are not included in this section. The interested reader may find such examples - and many others - in the test suite included in the appendix. Furthermore, only positive examples have been included here. In the discussion section, several examples for which the program accepts a non-ideal indexing are presented.

The examples from Anaphora and the Logic of Change

In Anaphora and the Logic of Change, Muskens used the example *"A farmer owns a donkey. The bastard beats it"*. Replicating the results, translating the sentence gives:

Ev1.Ev2.(((farmer v1) & ((donkey v2) & ((owns v2) v1))) & ((bastard v1) & ((beats v2) v1)))

Similarly, *"Every farmer who owns a donkey beats it"* is correctly translated as:

Av1.Av2.(~ ((farmer v1) & ((donkey v2) & ((owns v2) v1))) |((beats v2) v1))

Examples Illustrating Conjunctions

Demonstrating the effect of simple compositions using *and* (or .), the sentence *"A woman sleeps and a man sleeps"* gives:

Ev1.Ev2.(((woman v1) & (sleeps v1)) & ((man v2) & (sleeps v2)))

A similar sentence with universal quantifiers - *"every woman sleeps and every man sleeps"* - correctly translates as:

(Av1.(~(woman v1) |(sleeps v1)) & Av2.(~(man v2) |(sleeps v2)))

Illustrating how anaphora are picked up through conjunctions, the translation of *"a man loves a woman and the woman loves the man"* is:

Ev1.Ev2.(((man v1) & ((woman v2) & ((loves v2) v1))) & ((woman v2) & ((man v1) & ((loves v1) v2))))

Showcasing the new conjunction *so*, the sentence *"a farmer beats a donkey so the donkey hates the farmer"* translates as:

Av1.Av2.((~(farmer v1) |(~(donkey v2) |~((beats v2) v1))) |((donkey v2) & ((farmer v1) & ((hates v1) v2))))

To demonstrate the non-deterministic composition of programs $\cup$, consider the sentence *"a man sleeps or every man sleeps"*. As predicted through Θ, the sentence is correctly translated as:

(Ev1.((man v1) & (sleeps v1)) |Av2.(~(man v2) |(sleeps v2)))

Examples Illustrating Proper Nouns

Consider the sentences *"Carl hates bob. bob owns a donkey"* and *"Carl hates bob. carl owns a donkey"*. Due to the nominal filtering, they are correctly translated as:

Ev1.(((hates bob) carl) & ((donkey v1) & ((owns v1) bob)))

And:

Ev1.(((hates bob) carl) & ((donkey v1) & ((owns v1) carl)))

As a more complicated example of how proper nouns may act as both anaphora and generators, consider the sentence *"a man who loves alice owns a donkey so alice loves the donkey"*. The translation is:

Av1.Av2.(((~(man v1) |~((loves alice) v1)) |(~(donkey v2) |~((owns v2) v1))) |((donkey v2) & ((loves v2) alice)))

Examples Illustrating Adjectives and Being

Neatly demonstrating how entities are unified through the verb *is*, the sentence *"Bob is a farmer. he is good man* is correctly translated as:

((farmer bob) & ((man bob) & (good bob)))

Similarly, the (somewhat strange) sentences *"Alice loves a farmer. he is bob"* and *"Alice loves bob. he is a farmer"* are both correctly translated as:

((((man bob) & (good bob)) & ((loves bob) alice)))

Illustrating how the same unification holds under universal quantification, consider the sentence *"Every man who is a good farmer owns a donkey"*. The translation is:

Av1.(~((man v1) & ((farmer v1) & (good v1))) |Ev2.((donkey v2) & ((owns v2) v1)))

Similarly, *"bob beats no donkey so he is a good farmer"* gives:

(Ev1.((donkey v1) & ((beats v1) bob)) |((farmer bob) & (good bob)))

Chapter 8

Discussion

The presented strategy shows itself to be effective in translating sentences of limited length; except for a few edge cases addressed in the following sections, anaphora are correctly identified and mapped to entities. Furthermore, the combination of syntactic, nominal, and gender-based filtering has shown itself to be far more capable than initially expected, eliminating the immediate need for further filtering.

As the addition of the words *so* and *is*, and the linguistic class of adjectives show, the procedure can easily be expanded to include new definitions of words, more complex sentence structures, and even concepts such as encyclopaedic semantics.

Complexity-wise, the parsing algorithm is the definite bottleneck. Assume that s is a sentence with n words. Then, in order to parse s, a recursion tree with n levels is necessary. At a level m layers from the root, each node may have up to m children. As such, the complexity of the parser is $O(n \cdot (n-1) \cdot \ldots \cdot 1) = O(n!)$. In reality, backtracking is only necessary in the case of conjunctions. As such, if c is the number of conjunctions, only c branches of the parse tree are necessary. The entire sentence must still be traversed at each level, resulting in a decent complexity of $O(c! \cdot n^2)$.

8.1 Syntactic Ambiguities

Consider the sentence $s =$ *"A man sleeps and a woman sleeps or a donkey sleeps"*. Here, depending on the order in which the conjunctions are parsed, two candidates for $\psi(s)$ exists. The first proof is the following tree, corresponding to an translation of the sentence as $[\![s]\!] = \exists x_1 x_2 x_3((man\ x_1 \wedge sleeps\ x_1 \wedge woman\ x_2 \wedge sleeps\ x_2) \vee (donkey\ x_3 \wedge sleeps\ x_3))$:

$$\dfrac{\dfrac{\dfrac{or}{S\backslash(S/S)} \quad \dfrac{\dfrac{\dfrac{and}{S\backslash(S/S)} \quad \dfrac{\Psi(a\ man\ sleeps)}{S}}{S/S} \quad \dfrac{\Psi(a\ woman\ sleeps)}{S}}{S}}{S/S} \quad \dfrac{\Psi(a\ donkey\ sleeps)}{S}}{S}$$

Another possible translation for the sentence is $[\![s]\!] = \exists x_1 x_2 x_3((man\ x_1 \wedge sleeps\ x_1) \wedge ((woman\ x_2 \wedge sleeps\ x_2) \vee (donkey\ x_3 \wedge sleeps\ x_3)))$. The corresponding derivation tree is:

$$\dfrac{\dfrac{\dfrac{and}{S\backslash(S/S)} \quad \dfrac{\Psi(a\ man\ sleeps)}{S}}{S/S} \quad \dfrac{\dfrac{\dfrac{or}{S\backslash(S/S)} \quad \dfrac{\Psi(a\ woman\ sleeps)}{S}}{S\backslash S} \quad \dfrac{\Psi(a\ donkey\ sleeps)}{S}}{S}}{S}$$

At present, the program does nothing to address such ambiguities. A strategy based on either filtering or statistics should be implemented to alleviate this issue.

8.2 Logical Filtering

In some texts, references and referents can only be matched through logical means. Consider, for example, the myriad of jokes on the form:

> *"A_1 Dane, a_2 Swede, and a_3 Norwegian enter a bar. The_1 Swede says..."*

With slight modifications, the sentence becomes a member of the language of Δ:

"A Dane enters a bar. A Norwegian enters it. A Swede enters it. The Swede owns a donkey."

Translated, the sentence becomes:

Ev1.Ev2.Ev3.Ev4.Ev5.(((((dane v1) & ((bar v2) & ((enters v2) v1))) & ((norwegian v3) & ((enters v2) v3))) & ((swede v4) & ((enters v2) v4))) & ((swede v1) & ((donkey v5) & ((owns v5) v1))))

Hence, the program claims that v_1 is simultaneously Danish and Swedish!

In some cases, the information needed to glean the identity of a referent might lie with verb phrases and not noun phrases. Contrast the following two sentences:

"Carl$_1$ *poisoned Bob*$_2$*. He*$_2$ *died within a*$_3$ *week."*
"Carl$_1$ *poisoned Bob*$_2$*. He*$_1$ *was arrested within a*$_3$ *week."*

Rather than the aforementioned naive approach, it is likely that a solution can be found by inspecting the non-deictic content of the sentence. As such, some form of inference is necessary.

As pointed out by Allen [All95, p. 465-467], logical consistency is not the desired property - rather, a notion of **coherence** should be introduced. Several attempts at defining coherency are given, along with possible approaches to determine whether the property holds. [All95, p. 467-495]

Early in the design process, the intention was to include inference as the primary means of filtering. I imagined coupling the existing program with a deduction system based on the Isabelle theorem prover (see [PNW13]) and one of the strategies given by Allen.

Through empiric observation, I determined that - for the limited dictionary - filtering beyond name, gender, and syntax added little value. Furthermore, the examples where logical filtering is necessary mostly require encyclopaedic semantics to some extend. However, should the dictionary be expanded in that capacity, I believe it would prove highly interesting to explore logical filtering strategies.

8.3 A Little Help From Statistics

The primary purpose of this project is to translate and validate already paired sentences and coindexings. Generation of indexings is a secondary issue, and as such has only been superficially addressed.

Consider the relatively simple sentence *"a_1 man sleeps. a_2 farmer sleeps. he_x talks"*. There are two possible values for x. Both readings are semantically coherent. As such, the issue lies with the generation of indexings. The problem is solved through the intuition that a nearer generator ($x = 2$) is *more likely*. This suggests that a probability-based algorithm would be well-suited.

In recent years, machine learning methods has produced very successful solutions for generating indexings. Based on Expectation Maximization, coindexings can be generated with success rates in the vicinity of 80% [GHC98].

It is my opinion that a combination of a statistical generation-procedure and the logical translation- and validation-procedure could outperform either of the two alone, as the strengths of each approach counterbalance the weaknesses of the other.

Furthermore, while a combination of encyclopaedic semantics and logical filtering can to some extend address the problem of semantic ambiguity, syntactic ambiguity remains an issue. Utilizing statistical methods to choose between parse trees when multiple possibilities exist may lead to better results.

8.4 Other Improvements

Since the implemented algorithms serve merely to demonstrate a strategy, some are severely lacking in efficiency. Most glaring amongst these is the parser. As shown by Moot and Retoré, the *CYK algorithm* used for parsing context-free grammars can be adapted to produce an efficient parser for categorial grammars. [MR12]

Efficiency aside, improvements upon the strategy mainly expands in two directions: Additional words, and additional filters. Naturally, the two correlate - more complex languages require more complex filters. Should the dictionary expand to much larger size, it might be desirable to further study encyclopaedic semantics.

Immediately springing to mind are plurals, which in the way of filtering behave much like genders. Furthermore, plurals present an interesting opportunity to explore more complex anaphoric relationships. A similar case could be made for possessive pronouns, tenses and the passive form. Another avenue to explore is the notion of possible worlds, represented by words such as *believe*, *think* or *know*.

Finally, a strategy for resolving cataphoric references could be implemented - perhaps based on grammatical transformations, as presented by Chomsky. [Cho93]

Chapter 9

Conclusion

I have presented a formal logical method for semantic translation and anaphora resolution, largely based on the combination of Discourse Representation Theory and Montague Semantics applied by Reinhard Muskens in Anaphora and the Logic of Change.

I have designed a *proof-of-concept* implementation for the PolyML-language, demonstrating the theoretical groundwork. As discussed in the previous chapter, the resulting program performs well, although limited to a small set of the English sentences.

I have shown how the strategy can be expanded to include further definitions of syntactic classes, eventually covering a large set of the grammatical sentences of an ordinary language. For the best result, the method should be combined with a complementing statistical theory, such as those presented by Charniak and many others. Finally, I have illustrated several areas of interest into which future work could expand the strategy.

Appendix A

Source Code

```
(*-------------------------------------------------------------------*)
(* Definitions for various structures, constants, etc.:      *)
(*-------------------------------------------------------------------*)

(*Exceptions:*)
exception UnknownWord
exception UngrammaticalSentence
exception Type
exception WrongfulIndexing
exception DerivationException
exception RuntimeError
exception NoViableIndexing

(*Datatype declarations:*)
(*Categorial Grammar:*)
datatype Category =
     E
     | S
     | \ of Category*Category
     | / of Category*Category

infixr \;
infixr /;

(*Abbreviations*)
```

```
val VP = S / E;
val N = S \ E;
val NP = S / VP;

(*Static categories referenced by dictionary:*)
val noun = N;
val article = NP/N;
val verb_intransitive = VP;
val verb_monotransitive = VP/NP;
val proper_noun = NP;
val adjective = N/N
val pronoun = NP;
val conjunction = S\(S/S);

(*Logical Expressions:*)
datatype Term =
      Var of string
      | Abs of string*Term
      | App of Term*Term
      | Exists of string*Term
      | Forall of string*Term
      | Store of int*string
      | Conj of Term*Term
      | Eq of Term*Term
      | Differs of int list*string*string
      | Not of Term
      | Disj of Term*Term
      | Tr

(*Other datatypes:*)
datatype Token = T of string*Category*Term
datatype ParseTree =
      Fail
      | Leaf of Category*Token
      | Node of Category*ParseTree*ParseTree;


(*Counter for variable naming:*)
val counter = ref 0;
fun increment x = (counter := ((!counter) +x))

fun get_new_var () =
      let
          val void = increment 1
      in
          concat ["v", (Int.fmt (StringCvt.DEC) (!counter))]
      end
```

```
fun reset_var_counter () = (counter := 0)

(*---------------------------------------------------------------*)
(* Various generic functions:                                   *)
(*---------------------------------------------------------------*)
(*True if list contains s, false otherwise*)
fun contains [] s = false
  | contains (x::xs) s = (x=s) orelse (contains xs s)


fun print_term (Var(s)) = s
  | print_term (Tr) = "T"
  | print_term (Abs(s, t)) = concat ["L",s,".",(print_term t)]
  | print_term (App(t1, t2)) = concat ["(", (print_term t1), " ",
     (print_term t2), ")"]
  | print_term (Conj(t1, t2)) = concat ["(", (print_term t1), " & ",
     (print_term t2), ")"]
  | print_term (Disj(t1, t2)) = concat ["(", (print_term t1), " | ",
     (print_term t2), ")"]
  | print_term (Exists(s, t)) = concat ["E",s,".",(print_term t)]
  | print_term (Forall(s, t)) = concat ["A",s,".",(print_term t)]
  | print_term (Store(i, x)) = concat ["S(",(Int.fmt (StringCvt.DEC)
     i),",",x,")"]
  | print_term (Eq(t1, t2)) = concat ["(", (print_term t1), " = ",
     (print_term t2), ")"]
  | print_term (Differs(l, x1, x2)) = concat [x1,"[",print_list l,"]",
     x2]
  | print_term (Not(t)) = concat ["~", (print_term t)]
and print_list [] = ""
  | print_list [x] = (Int.fmt (StringCvt.DEC) x)
  | print_list (x::t) = concat [(Int.fmt (StringCvt.DEC) x), ",",
     print_list t]

(*Determines whether a given sentence is grammatical and converts it to
   a term:*)
fun to_term Fail = raise UngrammaticalSentence
  | to_term (Leaf(_, T(_, _, t))) = t
  | to_term (Node(_, t1, t2)) = App((to_term t1), (to_term t2))

(*Prints a derivation tree:*)
fun print_tree Fail = "fail"
  | print_tree (Leaf(_, T(s,_,_))) = s
  | print_tree (Node(_, t1, t2)) = concat ["(", print_tree t1, ",",
     print_tree t2, ")"];

(*Static terms referenced by dictionary*)
```

```
fun apply_content x =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
      in
          Abs(v1, Exists(v2, App(App(x, Var(v1)), Var(v2))))
      end

fun pronoun_term gender =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
          val v3 = get_new_var()
      in
          Abs(v1, Abs(v2, Abs(v3, Conj(App(gender, Store(0,
              v2)),App(App(App(Var(v1), Store(0, v2)), Var(v2)),
              Var(v3))))))
      end

fun verb_intransitive_term x =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
          val v3 = get_new_var()
      in
          Abs(v1, Abs(v2, Abs(v3, Conj(App(Var(x), Var(v1)), Eq(Var(v2),
              Var(v3))))))
      end

fun verb_monotransitive_term x =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
          val v3 = get_new_var()
          val v4 = get_new_var()
          val v5 = get_new_var()
      in
          Abs(v1,Abs(v2,App(Var(v1),Abs(v3,Abs(v4, Abs(v5,
              Conj(App(App(Var(x),Var(v3)),Var(v2)), Eq(Var(v4),
              Var(v5)))))))))
      end

fun proper_noun_term x gender =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
          val v3 = get_new_var()
```

```
      in
          Abs(v1, Abs(v2, Abs(v3, Conj(Eq(Store(0, v2), Var(x)),
             Conj(App(gender, Store(0, v2)),App(App(App(Var(v1),
             Store(0, v2)), Var(v2)), Var(v3)))))))
      end

fun noun_term x gender =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
          val v3 = get_new_var()
      in
          Abs(v1,Abs(v2,Abs(v3, Conj(App(gender, Var(v1)),
             Conj(App(Var(x), Var(v1)), Eq(Var(v2), Var(v3)))))))
      end

fun adjective_term x =
      let
          val v1 = get_new_var()
          val v2 = get_new_var()
          val v3 = get_new_var()
          val v4 = get_new_var()
      in
          Abs(v1, Abs(v2, Abs(v3, Abs(v4, Conj(App(App(App(Var(v1),
             Var(v2)), Var(v3)), Var(v4)), App(Var(x), Var(v2)))))))
      end
```

```
(*-------------------------------------------------------------------*)
(* Functions representing the main features of the program:     *)
(*-------------------------------------------------------------------*)

(*Import the other files:*)
use "Structures.ml";
use "Sets.ml";
use "Dictionary.ml";
use "Lexer.ml";
use "Parser.ml";
use "Binding.ml";
use "BetaReducer.ml";
use "Simplifier.ml";
use "GenderAndNameChecker.ml";
use "Indexer.ml";
use "Postprocessor.ml";

(*Translate a sentence with generated indexing:*)
fun translate s =
      let
```

```
        val void = reset_var_counter()
    in
        print_term (postprocess (translate_complete s))
    end

(*Translate a sentence with predetermined indexing:*)
fun translate_indexed s l =
    let
        val void = reset_var_counter()
    in
        print_term (postprocess (taut_replace (resolve_equalities
            (apply_to_final_term(taut_replace (extract_and_push
            (resolve_equalities (beta_reduce
            (apply_content(to_term(parse (apply_indexing l (lex
            s))))))))))))
    end;

(*Translate a sentence without applying simplifications:*)
fun translate_plain s l =
    let
        val void = reset_var_counter()
    in
        print_term (taut_replace (resolve_equalities(beta_reduce
            (apply_content(to_term(parse (apply_indexing l (lex
            s))))))))
    end;

(*Print the semantic representation of a single word:*)
fun meaning_of s = pr_to (tokenize s)
and pr_to (T(_,_,t)) = print_term t;

(*Import the test suite:*)
use "TestSuite.ml";
```

```
(*--------------------------------------------------------------*)
(* Functions for converting a string to a list of tokens:     *)
(*--------------------------------------------------------------*)

(*Read a list of characters, converting to strings:*)
(*Reading a space:*)
fun read_char_list s (#" "::t) =
    if s = [] then (read_char_list [] t)
    else (implode s)::(read_char_list [] t)

(*Reading a period:*)
  | read_char_list s (#"."::t) =
    let
```

```
          val rest = read_char_list [] t
      in
          if rest = [] then [implode s]
          else (implode s)::"and"::read_char_list [] t
      end
(*End of sentence:*)
  | read_char_list s [] =
      if s = [] then []
      else [implode s]

(*Reading a letter:*)
  | read_char_list s (x::t) = read_char_list (s @ [x]) t


(*Read a string, converting it to a list of words:*)
fun read_sentence s = read_char_list [] (explode s)

(*Convert a list of strings to a list of tokens:*)
fun tokenize_list [] = []
  | tokenize_list (x::xs) = (tokenize x)::(tokenize_list xs)

(*Read a string, converting it to a list of tokens:*)
fun lex s = tokenize_list (read_sentence s)
```

```
(*-------------------------------------------------------------------*)
(* Functions for building a derivation tree from a list of tokens. *)
(*                                                                 *)
(* Note: Rather inefficient brute-force strategy.                  *)
(*-------------------------------------------------------------------*)

(*Determine whether a derivation on two categories is possible:*)
fun deriv_cat_possible (a/b) c = if (b=c) then true else false
  | deriv_cat_possible c (b\a) = if (b=c) then true else false
  | deriv_cat_possible _ _ = false;

fun deriv_node_possible (Node(c1,_,_)) (Node(c2,_,_)) =
     deriv_cat_possible c1 c2
  | deriv_node_possible (Node(c1,_,_)) (Leaf(c2,_)) =
      deriv_cat_possible c1 c2
  | deriv_node_possible (Leaf(c1,_)) (Leaf(c2,_)) = deriv_cat_possible
      c1 c2
  | deriv_node_possible (Leaf(c1,_)) (Node(c2,_,_)) =
      deriv_cat_possible c1 c2
  | deriv_node_possible Fail _ = false
  | deriv_node_possible _ Fail = false

(*Determine the result of a derivation on two categories:*)
```

```
fun deriv_cat_result (a/b) (c\d) =
      if (b = (c\d)) then (a, false)
      else if (c = (a/b)) then (d, true)
           else raise DerivationException
  | deriv_cat_result (a/b) c = if (b=c) then (a, false) else raise
      DerivationException
  | deriv_cat_result c (b\a) = if (b=c) then (a, true) else raise
      DerivationException
  | deriv_cat_result _ _ = raise DerivationException;

fun deriv_node_result (Node(c1,t1,t2)) (Node(c2,t3,t4)) =
      let
          val (res, switch) = deriv_cat_result c1 c2
      in
          if switch then
              Node(res, (Node(c2,t3,t4)), (Node(c1,t1,t2)))
          else
              Node(res, (Node(c1,t1,t2)), (Node(c2,t3,t4)))
      end
  | deriv_node_result (Node(c1,t1,t2)) (Leaf(c2,t3)) =
      let
          val (res, switch) = deriv_cat_result c1 c2
      in
          if switch then
              Node(res, (Leaf(c2,t3)), (Node(c1,t1,t2)))
          else
              Node(res, (Node(c1,t1,t2)), (Leaf(c2,t3)))
      end
  | deriv_node_result (Leaf(c1,t1)) (Leaf(c2,t2)) =
      let
          val (res, switch) = deriv_cat_result c1 c2
      in
          if switch then
              Node(res, (Leaf(c2,t2)), (Leaf(c1,t1)))
          else
              Node(res, (Leaf(c1,t1)), (Leaf(c2,t2)))
      end
  | deriv_node_result (Leaf(c1,t1)) (Node(c2,t2,t3)) =
      let
          val (res, switch) = deriv_cat_result c1 c2
      in
          if switch then
              Node(res, (Node(c2,t2,t3)), (Leaf(c1,t1)))
          else
              Node(res, (Leaf(c1,t1)), (Node(c2,t2,t3)))
      end
```

```
  | deriv_node_result _ _ = raise DerivationException

(*Determine whether a sequence of trees has a derivation:*)
fun has_derivation [] = false
  | has_derivation [_] = false
  | has_derivation (x::(y::tail)) =
      if (deriv_node_possible x y) then true
      else has_derivation (y::tail)

(*Build a list of all possible derivations of some sequence of trees:*)
fun find_derivation_list _ [] = []
  | find_derivation_list _ [_] = []
  | find_derivation_list l (x::(y::tail)) =
      if (deriv_node_possible x y)
      then (l@(((deriv_node_result x y)::tail)))::
         (find_derivation_list (l@[x]) (y::tail))
      else find_derivation_list (l@[x]) (y::tail)


(*Build a parse tree from a sequence of subtrees:*)
fun build_tree [] = Fail
  | build_tree [Elem] = Elem
  | build_tree l =
      if (has_derivation l)
      then try_all (find_derivation_list [] l)
      else Fail
and try_all [] = Fail
  | try_all (x::xs) =
      let
         val res = build_tree x
      in
         if res = Fail then try_all xs else res
      end

(*Convert a list of tokens to a list of leaves containing those
   tokens:*)
fun to_leaf_list [] = []
  | to_leaf_list ((T(s,c,t))::xs) = (Leaf(c, T(s,c,t)))::(to_leaf_list
     xs)

fun parse l = build_tree (to_leaf_list l)
```

```
(*-----------------------------------------------------------*)
(* Functions for reducing a lambda-term through beta-reduction  *)
(* and alpha-conversion:                                     *)
(*-----------------------------------------------------------*)
```

```
(*Replace all instances of the variable s in t1 with t2:*)
fun replace (Var(x)) s t2 = if x = s then t2 else Var(x)
  | replace (App(te1, te2)) s t2 = App( (replace te1 s t2), (replace
     te2 s t2))
  | replace (Conj(te1, te2)) s t2 = Conj( (replace te1 s t2), (replace
     te2 s t2))
  | replace (Disj(te1, te2)) s t2 = Disj( (replace te1 s t2), (replace
     te2 s t2))
  | replace (Eq(te1, te2)) s t2 = Eq( (replace te1 s t2), (replace te2
     s t2))
  | replace (Exists(x, t)) s t2 = Exists(x, (replace t s t2))
  | replace (Abs(x, t)) s t2 = Abs(x, (replace t s t2))
  | replace (Forall(x, t)) s t2 = Forall(x, (replace t s t2))
  | replace (Store(i, x)) s (Var(s2)) =
      if (x = s)
      then Store(i, s2)
      else Store(i, x)
  | replace (Store(i, x)) s _ =
      if (x = s)
      then raise Type
      else Store(i, x)
  | replace (Not(t1)) s t2 = Not(replace t1 s t2)
  | replace (Differs(i2, x1, x2)) s (Var(s2)) =
      if s = x1 (*Note that x1 == x2 is a logical fallacy, should never
         happen*)
      then Differs(i2, s2, x2)
      else
          if s = x2
          then Differs(i2, x1, s2)
          else Differs(i2, x1, x2)
  | replace (Differs(i2, x1, x2)) s _ =
      if (x1 = s) orelse (x2 = s) then raise Type
      else (Differs(i2, x1, x2))
  | replace Tr _ _ = Tr


(*Recursively perform beta-reduction:*)
fun beta_reduce (App(Abs(s, t1), t2)) = beta_reduce (replace t1 s t2)
  | beta_reduce (App(t1, t2)) =
      let
          val res = beta_reduce t1
      in
          if res = t1
          then App(t1,t2)
          else beta_reduce (App(res, t2))
      end
  | beta_reduce (Abs(s, t)) = Abs(s, (beta_reduce t))
```

```
  | beta_reduce (Exists(s, t)) = Exists(s, (beta_reduce t))
  | beta_reduce (Forall(s, t)) = Forall(s, (beta_reduce t))
  | beta_reduce (Conj(t1,t2)) = Conj((beta_reduce t1), (beta_reduce t2))
  | beta_reduce (Disj(t1,t2)) = Disj((beta_reduce t1), (beta_reduce t2))
  | beta_reduce (Not(t)) = Not(beta_reduce t)
  | beta_reduce x = x;
```

```
(*-------------------------------------------------------------------*)
(* Functions for generating and applying an indexing:        *)
(*-------------------------------------------------------------------*)

(*Assigns store i to the term:*)
fun assign i (Store(_, s)) = Store(i,s)
  | assign i (Abs(s, t)) = Abs(s, (assign i t))
  | assign i (Exists(s, t)) = Exists(s, (assign i t))
  | assign i (Forall(s, t)) = Forall(s, (assign i t))
  | assign i (App(t1, t2)) = (App((assign i t1), (assign i t2)))
  | assign i (Conj(t1, t2)) = (Conj((assign i t1), (assign i t2)))
  | assign i (Var(s)) = Var(s)
  | assign i (Eq(t1, t2)) = (Eq((assign i t1), (assign i t2)))
  | assign i (Differs(_, s1, s2)) = Differs([i],s1, s2)
  | assign i (Not(t)) = Not(assign i t)
  | assign i (Disj(t1, t2)) = (Disj((assign i t1), (assign i t2)))
  | assign _ Tr = Tr

(*Determines whether the given category warrants an assignment:*)
fun should_assign x = (x = NP) orelse (x = (NP/N))

(*Applies an indexing on list format:*)
fun apply_indexing _ [] = []
  | apply_indexing [] ((T(s, c, t))::ys) =
      if (should_assign c)
      then  raise WrongfulIndexing
      else (T(s, c, t))::(apply_indexing [] ys)
  | apply_indexing (x::xs) ((T(s, c, t))::ys) =
      if (should_assign c)
      then (T(s, c, assign x t))::(apply_indexing xs ys)
      else (T(s, c, t))::(apply_indexing (x::xs) ys)




val current = ref []
val index_counter = ref 1

fun add_first l i max =
      if (contains l i) then
```

```
        let
            val void1 = current := ((!current)@[!index_counter])
            val void2 = index_counter := 1 + !index_counter
        in
            if i = max then ()
            else add_first l (i+1) max
        end
    else
        let
            val void = current := ((!current)@[1])
        in
            if i = max then ()
            else add_first l (i+1) max
        end

fun set_first i l =
    let
        val void1 = current := []
        val void2 = index_counter := 1
    in
        add_first l 1 i
    end

fun next_list i (h::t) l len =
    if (contains l len)
    then h::(next_list i t l (len+1))
    else
        if h = i
        then 1::(next_list i t l (len+1))
        else (h+1)::t
  | next_list _ [] _ _ = []

fun has_next_list i (x::xs) l len =
    if (i=1) then false
    else if (x <> i) andalso (not(contains l len)) then true
    else has_next_list i xs l (len+1)
  | has_next_list _ [] _ _ = false


fun recurse_generate e l=
    if (has_next_list e (!current) l 1)
    then
        let
            val void = current := (next_list e (!current) l 1)
        in
            (!current)::(recurse_generate e l)
        end
```

```
      else []


fun generate_possibilities indexes entities l =
      let
          val void = set_first indexes l
      in
          (!current)::(recurse_generate entities l)
      end

fun require_lock "a" = true
  | require_lock "every" = true
  | require_lock "no" = true
  | require_lock s = contains proper_noun_list s

fun new_index "a" = true
  | new_index "every" = true
  | new_index "no" = true
  | new_index "alice" = true
  | new_index "bob" = true
  | new_index "carl" = true
  | new_index "denise" = true
  | new_index "alex" = true
  | new_index "the" = true
  | new_index "he" = true
  | new_index "she" = true
  | new_index "it" = true
  | new_index "himself" = true
  | new_index "herself" = true
  | new_index "itself" = true
  | new_index "him" = true
  | new_index "her" = true
  | new_index _ = false;

fun get_indexes ((T(s,_,_))::xs) =
      if new_index s then 1 + (get_indexes xs)
      else (get_indexes xs)
  | get_indexes [] = 0

fun locked_indexes i ((T(s,_,_))::xs) =
      if require_lock s then i::(locked_indexes (i+1) xs)
      else
          if new_index s then (locked_indexes (i+1) xs)
          else (locked_indexes (i) xs)

  | locked_indexes i [] = []
```

```
fun try_translate _ [] = raise NoViableIndexing
  | try_translate l (x::xs) =
      let
          val tree = parse (apply_indexing x l)
      in
          if (government_binding tree) then
          let
              val fin = simplify (beta_reduce (apply_content(to_term
                 tree)))
          in
              if (name_check fin) andalso (gender_check fin) then fin
              else try_translate l xs
          end
          else try_translate l xs
      end

fun translate_complete s =
      let
          val token_list = (lex s)
          val i = get_indexes token_list
      in
          (try_translate token_list (generate_possibilities i i
             (locked_indexes 1 token_list) ))
      end
```

```
(*--------------------------------------------------------------*)
(* Functions for pushing negations downwards and extracting    *)
(* quantors, putting an expression on negation normal form:    *)
(*--------------------------------------------------------------*)

(*Negation not present:*)
fun push_negations (Abs(s,x)) = Abs(s, push_negations x)
  | push_negations (Conj(t1,t2)) = Conj(push_negations
     t1,push_negations t2)
  | push_negations (Disj(t1,t2)) = Disj(push_negations
     t1,push_negations t2)
  | push_negations (Exists(s,x)) = Exists(s, push_negations x)
  | push_negations (Forall(s,x)) = Forall(s, push_negations x)

(*Double negation cancelled:*)
  | push_negations (Not(Not(t))) = push_negations t

(*Negated quantors:*)
  | push_negations (Not(Exists(s,x))) = Forall(s, push_negations
     (Not(x)))
  | push_negations (Not(Forall(s,x))) = Exists(s, push_negations
     (Not(x)))
```

```
(*De Morgan's Laws:*)
  | push_negations (Not((Disj(t1,t2)))) = Conj(push_negations
     (Not(t1)),push_negations (Not(t2)))
  | push_negations (Not((Conj(t1,t2)))) = Disj(push_negations
     (Not(t1)),push_negations (Not(t2)))

(*Other cases:*)
  | push_negations x = x

(* Finds a removes the next quantifier.
** Returns a triple consisting of a number representing whether it is
** existential or universal, the name of the variable,
** and the resulting term:*)
fun next_quantifier (Exists(s, t)) = (1, s, t)
  | next_quantifier (Forall(s, t)) = (2, s, t)
  | next_quantifier (Not(t)) =
      let
          val (x,y,z) = next_quantifier t
      in
          (x,y, Not(z))
      end
  | next_quantifier (Conj(t1,t2)) =
      let
          val (x1,y1,z1) = next_quantifier t1
      in
          (*Try to find quantifier in lhs:*)
          if (x1 <> 0)
          then (x1, y1, Conj(z1, t2))
          else
              (*Try to find quantifier in rhs:*)
              let
                  val (x2, y2, z2) = next_quantifier t2
              in
                  (x2, y2, Conj(t1, z2))
              end
      end
  | next_quantifier (Disj(t1,t2)) =
      let
          val (x1,y1,z1) = next_quantifier t1
      in
          (*Try to find quantifier in lhs:*)
          if (x1 <> 0)
          then (x1, y1, Disj(z1, t2))
          else
              (*Try to find quantifier in rhs:*)
              let
```

```
                val (x2, y2, z2) = next_quantifier t2
            in
                (x2, y2, Disj(t1, z2))
            end
        end
  | next_quantifier x = (0, "", x)

  (*Checks if a term contains a quantifier:*)
fun has_next_quantifier (Forall(_,_)) = true
  | has_next_quantifier (Exists(_,_)) = true
  | has_next_quantifier (Conj(t1, t2)) = (has_next_quantifier t1)
     orelse (has_next_quantifier t2)
  | has_next_quantifier (Disj(t1, t2)) = (has_next_quantifier t1)
     orelse (has_next_quantifier t2)
  | has_next_quantifier (Not(t)) = (has_next_quantifier t)
  | has_next_quantifier _ = false

(*Extract all quantifiers to the topmost level of a term:*)
fun extract_quantifiers t =
      (*Do nothing if the term contains no quantifiers:*)
      if (not(has_next_quantifier t)) then t
      else
         let
            val (typ, var, res) = next_quantifier t
         in
            if (typ = 1) then Exists(var, extract_quantifiers res)
            else if (typ = 2) then Forall(var, extract_quantifiers res)
                else raise RuntimeError
         end

(*Puts a term on negation normal form:*)
fun nnf (Abs(s,t)) = Abs(s, (push_negations t))

(*------------------------------------------------------------------*)
(* Resolve variable equalities within a term:                     *)
(*------------------------------------------------------------------*)

(*Find the set of variables equal to x within a term:*)
fun find_equality x (Eq(Var(s1), Var(s2))) =
      if ((x = s1) andalso (s1 <> s2)) then singleton s2 else
         if ((x = s2) andalso (s1 <> s2)) then singleton s1 else
            empty_set
  | find_equality x (Conj(t1, t2)) = set_union (find_equality x t1)
      (find_equality x t2)
  | find_equality x (Exists(_, t)) = find_equality x t
  | find_equality x (Forall(_, t)) = find_equality x t
  | find_equality x (Disj((Not(t1)), t2)) = set_union (find_equality x
```

```
    t1) (find_equality x t2)
  | find_equality x (Disj(t1, Not(t2))) = set_union (find_equality x
    t1) (find_equality x t2)
  | find_equality x (Disj(t1, t2)) = set_union (find_equality x t1)
    (find_equality x t2)
  | find_equality _ _ = empty_set

fun replace_all (x::xs) v s = replace v s (Var(x))
  | replace_all [] v s = v

(*Resolve state equalities:*)
fun resolve_equalities (Abs(s, t)) = Abs(s, resolve_equalities t)
  | resolve_equalities (Exists(s, t)) =
      let
          val v = resolve_equalities t
          val q = find_equality s v
      in
          if q <> empty_set then (replace_all q v s )
          else Exists(s,v)
      end
  | resolve_equalities (Forall(s, t)) =
      let
          val v = resolve_equalities t
          val q = find_equality s v
      in
          if q <> empty_set then (replace_all q v s )
          else Forall(s,v)
      end
  | resolve_equalities (Disj(t1, t2)) = Disj(resolve_equalities t1,
    resolve_equalities t2)
  | resolve_equalities (Conj(t1, t2)) = Conj(resolve_equalities t1,
    resolve_equalities t2)
  | resolve_equalities (Not(t)) = Not(resolve_equalities t)
  | resolve_equalities x = x;

(*--------------------------------------------------------------*)
(* Remove tautological statements from expressions:            *)
(*--------------------------------------------------------------*)

(*Replaces tautological expressions with Tr:*)
fun taut_replace (Eq(t1, t2)) = if t1 = t2 then Tr else (Eq(t1, t2))
  | taut_replace (Conj(t1, t2)) =
      let
          val left = taut_replace t1
          val right = taut_replace t2
      in
          if left = Tr then right
```

```
        else if right = Tr then left
             else if (left = Not(Tr)) orelse (right = Not(Tr))
                  then Not(Tr)
                  else (Conj(left, right))
    end
  | taut_replace (Disj(t1, t2)) =
    let
        val left = taut_replace t1
        val right = taut_replace t2
    in
        if left = Tr orelse right = Tr then Tr
        else if left = Not(Tr) then right
             else if right = Not(Tr) then left
                  else Disj(left, right)
    end
  | taut_replace (Not(t)) =
    let
        val inner = taut_replace t
    in
        if (inner = Not(Tr)) then Tr
        else Not(inner)
    end
  | taut_replace (Exists(s,t)) = Exists(s, taut_replace t)
  | taut_replace (Forall(s,t)) = Forall(s, taut_replace t)
  | taut_replace (Abs(s,t))   =  Abs(s, taut_replace t)
  | taut_replace x = x

(*-----------------------------------------------------------------*)
(* Set a term up for Unselective Binding Lemma application:   *)
(*-----------------------------------------------------------------*)

(*Returns the next existential successor to a variable:*)
fun existential_successor (Differs([i], s1, s2)) var = if s1 = var then
    (i, s2) else (0,"")
  | existential_successor (Conj(t1,t2)) var =
    let
        val left = existential_successor t1 var
    in
        if left <> (0, "") then left
        else existential_successor t2 var
    end
  | existential_successor (Disj(t1,t2)) var =
    let
        val left = existential_successor t1 var
        val right = existential_successor t2 var
    in
        if left <> (0,"") andalso left = right then left
```

```
        else (0, "")
      end
  | existential_successor (Exists(s, t)) var = existential_successor t
     var
  | existential_successor (Forall(s, t)) var = existential_successor t
     var
  | existential_successor _ _ = (0, "")

(*Returns the next universal successor to a variable:*)
fun universal_successor (Not(Differs([i], s1, s2))) var = if s1 = var
   then (i, s2) else (0,"")
  | universal_successor (Conj(t1,t2)) var =
      let
         val left = universal_successor t1 var
         val right = universal_successor t1 var
      in
         if left <> (0,"") andalso left = right then left
         else (0, "")
      end
  | universal_successor (Disj(t1,t2)) var =
      let
         val left = universal_successor t1 var
      in
         if left <> (0, "") then left
         else universal_successor t2 var
      end
  | universal_successor (Exists(s, t)) var = universal_successor t var
  | universal_successor (Forall(s, t)) var = universal_successor t var
  | universal_successor _ _ = (0, "")

(*Removes the first part of every tuple in a list:*)
fun purge_first [] = []
  | purge_first ((x,y)::t) = y::(purge_first t)

(*Removes the second part of every tuple in a list:*)
fun purge_second [] = []
  | purge_second ((x,y)::t) = x::(purge_second t)

(*Get the last element of a list:*)
fun get_last [] = raise RuntimeError
  | get_last [x] = x
  | get_last (_::xs) = get_last xs

(*Get all elements but the last of a list:*)
fun all_but_last [] = raise RuntimeError
  | all_but_last [x] = []
  | all_but_last (x::xs) = x::(all_but_last xs)
```

```
(*Removes all quantifiers contained in a list from an expression:*)
fun remove_quantifiers (Exists(s, t)) l =
      if contains l s then remove_quantifiers t l
      else Exists(s, remove_quantifiers t l)
  | remove_quantifiers (Forall(s, t)) l =
      if contains l s then remove_quantifiers t l
      else Forall(s, remove_quantifiers t l)
  | remove_quantifiers (Conj(t1, t2)) l = Conj(remove_quantifiers t1 l,
     remove_quantifiers t2 l)
  | remove_quantifiers (Disj(t1, t2)) l = Disj(remove_quantifiers t1 l,
     remove_quantifiers t2 l)
  | remove_quantifiers (Not(t)) l = Not(remove_quantifiers t l)
  | remove_quantifiers x _ = x

(*Replace all difference-terms concerning the given list of states with
   true:*)
fun remove_diff (Differs(i, s1, s2)) l = if (contains l s2) then Tr
   else (Differs(i, s1, s2))
  | remove_diff (Conj(t1, t2)) l = (Conj(remove_diff t1 l, remove_diff
     t2 l))
  | remove_diff (Disj(t1, t2)) l = (Disj(remove_diff t1 l, remove_diff
     t2 l))
  | remove_diff (Exists(s,t)) l = Exists(s, remove_diff t l)
  | remove_diff (Forall(s,t)) l = Forall(s, remove_diff t l)
  | remove_diff (Not(t)) l = Not(remove_diff t l)
  | remove_diff x _ = x

(*True if i occurs before s in a list of pairs:*)
fun earlier i1 s1 ((i2, s2)::tail) =
      if i1 = i2 then true
      else if s1 = s2 then false
          else earlier i1 s1 tail
  | earlier _ _ [] = false

(*True if s1 occurs as a secondary item in a pair in a list*)
fun contains_state s1 ((_, s2)::tail) = if s1 = s2 then true else
   contains_state s1 tail
  | contains_state _ [] = false

fun first_state ((_, s)::_) = s
  | first_state [] = raise RuntimeError

fun last_state [(_, s)] = s
  | last_state (x::xs) = last_state xs
  | last_state [] = raise RuntimeError
```

```
(*Push stores forward or backward:*)
fun push_stores (Store(i, s)) mapping top_var =
      if (contains_state s mapping) then
          if (earlier i s mapping) then (Store(i, last_state mapping))
          else (Store(i, top_var))
      else (Store(i, s))
  | push_stores (Differs(l, s1, s2)) mapping top_var =
      if (contains_state s1 mapping) then Differs(l, last_state
         mapping, s2)
      else (Differs(l, s1, s2))
  | push_stores (Conj(t1, t2)) mapping top_var = (Conj(push_stores t1
     mapping top_var, push_stores t2 mapping top_var))
  | push_stores (App(t1, t2)) mapping top_var = (App(push_stores t1
     mapping top_var, push_stores t2 mapping top_var))
  | push_stores (Exists(s,t)) mapping top_var = Exists(s, push_stores t
     mapping top_var)
  | push_stores (Forall(s,t)) mapping top_var = Forall(s, push_stores t
     mapping top_var)
  | push_stores (Disj(t1, t2)) mapping top_var = (Disj(push_stores t1
     mapping top_var, push_stores t2 mapping top_var))
  | push_stores (Not(t)) mapping top_var = Not(push_stores t mapping
     top_var)
  | push_stores (Eq(t1, t2)) mapping top_var = (Eq(push_stores t1
     mapping top_var, push_stores t2 mapping top_var))
  | push_stores x _ _ = x


(*Returns true if the given variable is existentially quantified:*)
fun exist_quant s (Exists(s2, t)) = if s = s2 then true else
    exist_quant s t
  | exist_quant s (Forall(s2, t)) = exist_quant s t
  | exist_quant s (Conj(t1,t2)) = (exist_quant s t1) orelse
     (exist_quant s t2)
  | exist_quant s (Disj(t1,t2)) = (exist_quant s t1) orelse
     (exist_quant s t2)
  | exist_quant _ _ = false

(*Returns true if the given variable is universally quantified:*)
fun forall_quant s (Forall(s2, t)) = if s = s2 then true else
    forall_quant s t
  | forall_quant s (Exists(s2, t)) = forall_quant s t
  | forall_quant s (Conj(t1,t2)) = (forall_quant s t1) orelse
     (forall_quant s t2)
  | forall_quant s (Disj(t1,t2)) = (forall_quant s t1) orelse
     (forall_quant s t2)
  | forall_quant _ _ = false
```

```
(*Determine existential and universal successors to a state:*)
fun existential_successor_list term var used_stores =
      let
          val (store, state) = existential_successor term var
      in
          if (store <> 0) andalso (exist_quant state term) andalso
             (not(contains used_stores store))
          then (store, state)::(existential_successor_list term state
             (store::used_stores))
          else []
      end

fun universal_successor_list term var used_stores =
      let
          val (store, state) = universal_successor term var
      in
          if (store <> 0) andalso (forall_quant state term) andalso
             (not(contains used_stores store))
          then (store, state)::(universal_successor_list term state
             (store::used_stores))
          else []
      end


(*Extract state information, push stores:*)
fun extract_and_push_recurse term top_var =
      let
          val mapping = existential_successor_list term top_var []
      in
          (*If an existential successor exists:*)
          if mapping <> [] then
              let
                  val state_list = purge_first mapping
                  val last = last_state mapping
                  val after_push = push_stores term mapping top_var
                  val rest = extract_and_push_recurse
                      (remove_quantifiers (remove_diff after_push
                      state_list) state_list) last
              in
                  Exists(last, Conj(Differs(purge_second mapping,
                      top_var, last), rest))
              end

          (*If a universal successor exists:*)
          else
              let
```

```
                val mapping = universal_successor_list term top_var []
            in
                if mapping <> [] then
                    let
                        val state_list = purge_first mapping
                        val last = last_state mapping
                        val after_push = push_stores term mapping
                            top_var
                        val rest = extract_and_push_recurse
                            (remove_quantifiers (remove_diff after_push
                            state_list) state_list) last
                    in
                        Forall(last, Disj(Not(Differs(purge_second
                            mapping, top_var, last)), rest))
                    end

                (*If no successor can be found yet:*)
                else
                    continue term top_var
            end
    end

and continue (Conj(t1, t2)) top_var = Conj(extract_and_push_recurse t1
    top_var, extract_and_push_recurse t2 top_var)
  | continue (Disj(t1, t2)) top_var = Disj(extract_and_push_recurse t1
      top_var, extract_and_push_recurse t2 top_var)
  | continue (Not(t)) top_var = Not(extract_and_push_recurse t top_var)
  | continue (Exists(s,t)) top_var = Exists(s, extract_and_push_recurse
      t top_var)
  | continue (Forall(s,t)) top_var = Forall(s, extract_and_push_recurse
      t top_var)
  | continue x top_var = x

fun extract_and_push (Abs(s,t)) = Abs(s, (extract_and_push_recurse t s))

(*---------------------------------------------------------------------*)
(* Apply Unselective Binding Lemma to a term:                         *)
(*---------------------------------------------------------------------*)

(*Generates a list of new variables from a list of integers:*)
fun generate_var_list [] = []
  | generate_var_list (x::xs) = (x, get_new_var())::(generate_var_list
      xs)

(*Checks if a term is a difference-expression. Returns list of new
    variables if yes.*)
fun gather_vars (Differs(i, s1, s2)) s4 =
```

```
      if (s2 = s4) then generate_var_list i
      else []
  | gather_vars _ _ = []

(*Adds a bunch of existential quantifiers to a term:*)
fun add_existential_quantifiers t (x::xs) = Exists(x,
    (add_existential_quantifiers t xs))
  | add_existential_quantifiers t [] = t

(*Adds a bunch of universal quantifiers to a term:*)
fun add_universal_quantifiers t (x::xs) = Forall(x,
    (add_universal_quantifiers t xs))
  | add_universal_quantifiers t [] = t


(*Find the variable corresponding to a given integer in a list. Raises
    exception if not found.*)
fun find_corresponding _ [] = "WHY"
  | find_corresponding i ((x1, x2)::xs) = if i = x1 then x2 else
     find_corresponding i xs

(*Replaces all store references corresponding to a state with variables
    from a list:*)
fun replace_stores (Store(i, s1)) s2 l =
      if s1 = s2 then Var(find_corresponding i l)
      else (Store(i, s1))
  | replace_stores (Conj(t1,t2)) s l = (Conj(replace_stores t1 s
     l,replace_stores t2 s l))
  | replace_stores (Disj(t1,t2)) s l = (Disj(replace_stores t1 s
     l,replace_stores t2 s l))
  | replace_stores (Not(t1)) s l = (Not(replace_stores t1 s l))
  | replace_stores (App(t1,t2)) s l = (App(replace_stores t1 s
     l,replace_stores t2 s l))
  | replace_stores (Eq(t1,t2)) s l = (Eq(replace_stores t1 s
     l,replace_stores t2 s l))
  | replace_stores (Exists(s1,t)) s l = (Exists(s1, replace_stores t s
     l))
  | replace_stores (Forall(s1,t)) s l = (Forall(s1, replace_stores t s
     l))
  | replace_stores x _ _ = x

(*Apply unselective binding lemma to a term in PLs:*)
fun apply_binding_lemma (Exists(s,Conj(t1, t2))) =
      let
          val store_list = gather_vars t1 s
      in
          if store_list <> []
```

```
        then add_existential_quantifiers (apply_binding_lemma
           (replace_stores t2 s (store_list)) ) (purge_first
           store_list)
        else (Exists(s,Conj(t1, apply_binding_lemma t2 )))
     end

  | apply_binding_lemma (Forall(s,Disj(Not(t1), t2))) =
     let
        val store_list = gather_vars t1 s
     in
        if store_list <> []
        then add_universal_quantifiers (apply_binding_lemma
           (replace_stores t2 s (store_list)) ) (purge_first
           store_list)
        else (Forall(s,Disj(Not(t1), apply_binding_lemma t2 )))
     end
  | apply_binding_lemma (Conj(t1,t2)) = (Conj(apply_binding_lemma t1 ,
     apply_binding_lemma t2 ))
  | apply_binding_lemma (Disj(t1,t2)) = (Disj(apply_binding_lemma t1 ,
     apply_binding_lemma t2 ))
  | apply_binding_lemma (Not(t)) = (Not(apply_binding_lemma t ))
  | apply_binding_lemma x = x


fun has_corresponding [] _ = false
  | has_corresponding ((x,_)::xs) (y,z) = (x = y) orelse
     (has_corresponding xs (y,z))

fun merge [] l = l
  | merge (x::xs) l = if has_corresponding l x then merge xs l else
     merge xs (x::l)

(*Gather the names and numbers of all stores used in the initial
    state:*)
fun gather_initial_stores s (Store(i, s2)) = if s = s2 then [(i,
    get_new_var())] else []
  | gather_initial_stores s (Conj(t1, t2)) = merge
     (gather_initial_stores s t1) (gather_initial_stores s t2)
  | gather_initial_stores s (Disj(t1, t2)) = merge
     (gather_initial_stores s t1) (gather_initial_stores s t2)
  | gather_initial_stores s (App(t1, t2)) = merge
     (gather_initial_stores s t1) (gather_initial_stores s t2)
  | gather_initial_stores s (Eq(t1, t2)) = merge (gather_initial_stores
     s t1) (gather_initial_stores s t2)
  | gather_initial_stores s (Not(t)) = gather_initial_stores s t
  | gather_initial_stores s (Exists(_,t)) = gather_initial_stores s t
  | gather_initial_stores s (Forall(_,t)) = gather_initial_stores s t
```

```
  | gather_initial_stores s _ = []

(*Apply unselective binding lemma to a term of the type s -> t. Will
   produce non-deictic content.*)
fun apply_to_final_term (Abs(s, t)) =
      let
         val store_list = gather_initial_stores s t
      in
         if store_list <> []
         then add_existential_quantifiers (apply_binding_lemma
             (replace_stores t s (store_list)) ) (purge_first
             store_list)
         else apply_binding_lemma t
      end

fun simplify t = (taut_replace (resolve_equalities
    (apply_to_final_term(taut_replace (extract_and_push(nnf
    (resolve_equalities t)))))))
```

```
(*------------------------------------------------------------------*)
(* Functions implementing filters on gender and name:         *)
(*------------------------------------------------------------------*)

(*Recursively check that no proper noun is used to refer to two
   different entities:*)
fun name_check_inner quant_var (Eq(Var(v1), Var(v2))) = if
    (not(contains quant_var v1)) andalso (not(contains quant_var v2))
    then false else true
  | name_check_inner quant_var (Exists(s, t)) = name_check_inner
     (s::quant_var) t
  | name_check_inner quant_var (Forall(s, t)) = name_check_inner
     (s::quant_var) t
  | name_check_inner quant_var (Abs(s, t)) = name_check_inner
     (s::quant_var) t
  | name_check_inner quant_var (Disj(t1, t2)) = (name_check_inner
     quant_var t1) andalso (name_check_inner quant_var t2)
  | name_check_inner quant_var (Conj(t1, t2)) = (name_check_inner
     quant_var t1) andalso (name_check_inner quant_var t2)
  | name_check_inner _ _ = true

fun name_check t = name_check_inner [] t;

(*Recursively check that genders are consistent in the current
   interpretation:*)
fun gender_check_inner s (App(Var(x), Var(s2))) =
      if (s = s2) andalso (((x = "male") orelse (x = "female")) orelse
          (x = "neuter"))
```

```
      then singleton x
      else ["male", "female", "neuter"]
  | gender_check_inner s (App(Not(Var(x)), Var(s2))) =
      if (s = s2) andalso (((x = "male") orelse (x = "female")) orelse
         (x = "neuter"))
      then set_minus ["male", "female", "neuter"] (singleton x)
      else ["male", "female", "neuter"]
  | gender_check_inner s (Conj(t1, t2)) = set_intersection
     (gender_check_inner s t1) (gender_check_inner s t2)
  | gender_check_inner s (Disj(t1, t2)) = set_intersection
     (gender_check_inner s t1) (gender_check_inner s t2)
  | gender_check_inner s (Not(t)) = (gender_check_inner s t)
  | gender_check_inner s (Exists(_,t)) = (gender_check_inner s t)
  | gender_check_inner s (Forall(_,t)) = (gender_check_inner s t)
  | gender_check_inner s (Abs(_,t)) = (gender_check_inner s t)
  | gender_check_inner s _ = ["male", "female", "neuter"]


fun gender_check_traverse (Exists(s, t)) =
      let
         val res = gender_check_inner s t
      in
         if res = empty_set then false
         else gender_check_traverse t
      end
  | gender_check_traverse (Forall(s, t)) =
      let
         val res = gender_check_inner s t
      in
         if res = empty_set then false
         else gender_check_traverse t
      end
  | gender_check_traverse (Conj(t1, t2)) = (gender_check_traverse t1)
     andalso (gender_check_traverse t2)
  | gender_check_traverse (Disj(t1, t2)) = (gender_check_traverse t1)
     andalso (gender_check_traverse t2)
  | gender_check_traverse (Not(t1)) = (gender_check_traverse t1)
  | gender_check_traverse _ = true;

fun gender_check_proper_nouns [] t = true
  | gender_check_proper_nouns (x::xs) t = ((gender_check_inner x t) <>
     empty_set) andalso (gender_check_proper_nouns xs t)

fun gender_check t = (gender_check_proper_nouns proper_noun_list t)
   andalso (gender_check_traverse t)
```

Appendix B

Auxiliary Code

```
(*------------------------------------------------------------------*)
(* Implements sets and various pertaining functions:          *)
(*------------------------------------------------------------------*)

(*Defining the datatype:*)
datatype 'a set = list of 'a;

val empty_set = []
fun singleton x = [x]

(*Looking up: *)
fun set_contains (h::t) elem = (h = elem) orelse (set_contains t elem)
  | set_contains [] elem = false

(*Adding an element:*)
fun set_add s elem =
     if (set_contains s elem) then s
     else elem :: s

(*Removing an element:*)
fun set_minus [] _ = []
  | set_minus (h::t) s2 =
     if set_contains s2 h then set_minus t s2
     else h::(set_minus t s2)
```

```
(*Operations between sets:*)
fun set_union (h::t) s2 = set_add (set_union t s2) h
  | set_union [] s2 = s2

fun set_intersection (h::t) s2 =
      if set_contains s2 h then h::(set_intersection t s2)
      else (set_intersection t s2)
  | set_intersection [] s2 = []
```

Appendix C

Sample Dictionary

```
(*-----------------------------------------------------------------*)
(* Function defining the dictionary.                        *)
(*                                                          *)
(* Note: Genders simplified for efficiency.                 *)
(*-----------------------------------------------------------------*)

(*Pronouns:*)
fun tokenize "he" = T("he", pronoun, (pronoun_term (Var("male"))))
  | tokenize "she" = T("she", pronoun, (pronoun_term (Var("female"))))
  | tokenize "him" = T("him", pronoun, (pronoun_term (Var("male"))))
  | tokenize "her" = T("her", pronoun, (pronoun_term (Var("female"))))
  | tokenize "it" = T("it", pronoun, (pronoun_term (Var("neuter"))))
  | tokenize "himself" = T("himself", pronoun, (pronoun_term
     (Var("male"))))
  | tokenize "herself" = T("herself", pronoun, (pronoun_term
     (Var("female"))))
  | tokenize "itself" = T("itself", pronoun, (pronoun_term
     (Var("neuter"))))

(*Intransitive verbs:*)
  | tokenize "sleeps" = T("sleeps", verb_intransitive,
     verb_intransitive_term "sleeps")
  | tokenize "walks" = T("walks", verb_intransitive,
     verb_intransitive_term "walks")
```

```
  | tokenize "talks" = T("talks", verb_intransitive,
     verb_intransitive_term "talks")

(*Monotransitive verbs:*)
  | tokenize "loves" = T("loves", verb_monotransitive,
     verb_monotransitive_term "loves")
  | tokenize "hates" = T("hates", verb_monotransitive,
     verb_monotransitive_term "hates")
  | tokenize "beats" = T("beats", verb_monotransitive,
     verb_monotransitive_term "beats")
  | tokenize "owns" = T("owns", verb_monotransitive,
     verb_monotransitive_term "owns")
  | tokenize "knows" = T("knows", verb_monotransitive,
     verb_monotransitive_term "knows")
  | tokenize "enters" = T("enters", verb_monotransitive,
     verb_monotransitive_term "enters")

(*Proper nouns:*)
  | tokenize "alice" = T("alice", proper_noun, (proper_noun_term
     "alice" (Var("female"))))
  | tokenize "bob" = T("bob", proper_noun, (proper_noun_term "bob"
     (Var("male"))))
  | tokenize "carl" = T("carl", proper_noun, (proper_noun_term "carl"
     (Var("male"))))
  | tokenize "denise" = T("denise", proper_noun, (proper_noun_term
     "denise" (Var("female"))))
  | tokenize "alex" = T("alex", proper_noun, (proper_noun_term "alex"
     (Not(Var("neuter")))))

(*Common nouns:*)
  | tokenize "man" = T("man", noun, (noun_term "man" (Var("male"))))
  | tokenize "woman" = T("woman", noun, (noun_term "woman"
     (Var("female"))))
  | tokenize "farmer" = T("farmer", noun, (noun_term "farmer"
     (Not(Var("neuter")))))
  | tokenize "donkey" = T("donkey", noun, (noun_term "donkey"
     (Var("neuter"))))
  | tokenize "bastard" = T("bastard", noun, (noun_term "bastard"
     (Abs(get_new_var(), Tr))))
  | tokenize "animal" = T("animal", noun, (noun_term "animal"
     (Var("neuter"))))
  | tokenize "dane" = T("dane", noun, (noun_term "dane"
     (Not(Var("neuter")))))
  | tokenize "norwegian" = T("norwegian", noun, (noun_term "norwegian"
     (Not(Var("neuter")))))
  | tokenize "swede" = T("swede", noun, (noun_term "swede"
     (Not(Var("neuter")))))
```

```
  | tokenize "bar" = T("bar", noun, (noun_term "bar" (Var("neuter"))))

(*Adjectives:*)
  | tokenize "good" = T("good", adjective, adjective_term("good"))
  | tokenize "bad" = T("bad", adjective, adjective_term("bad"))
  | tokenize "poor" = T("poor", adjective, adjective_term("poor"))
  | tokenize "lazy" = T("lazy", adjective, adjective_term("lazy"))

(*Articles:*)
  | tokenize "a" = T("a",article,
     let
         val v1 = get_new_var()
         val v2 = get_new_var()
         val v3 = get_new_var()
         val v4 = get_new_var()
         val v5 = get_new_var()
         val v6 = get_new_var()
     in
         Abs(v1, Abs(v2,Abs(v3,Abs(v4, Exists(v5,
            Exists(v6,Conj(Differs([], v3, v5)
            ,Conj(App(App(App(Var(v1), Store(0, v5)), Var(v5)),
            Var(v6)), App(App(App(Var(v2), Store(0, v5)), Var(v6)),
            Var(v4)))))))))
     end
    )
  | tokenize "the" = T("the",article,
     let
         val v1 = get_new_var()
         val v2 = get_new_var()
         val v3 = get_new_var()
         val v4 = get_new_var()
         val v5 = get_new_var()
     in
         Abs(v1, Abs(v2,Abs(v3,Abs(v4, Exists(v5,
            Conj(App(App(App(Var(v1), Store(0, v5)), Var(v3)),
            Var(v5)), App(App(App(Var(v2), Store(0, v5)), Var(v5)),
            Var(v4))))))))
     end
    )
  | tokenize "every" = T("every",article,
     let
         val v1 = get_new_var()
         val v2 = get_new_var()
         val v3 = get_new_var()
         val v4 = get_new_var()
         val v5 = get_new_var()
         val v6 = get_new_var()
```

```
            val v7 = get_new_var()
        in
            Abs(v1, Abs(v2, Abs(v3, Abs(v4, Conj(Eq(Var(v3), Var(v4)),
               Forall(v5, Forall(v6, Disj(Not(Conj(Differs([], v3, v5),
               App(App(App(Var(v1), Store(0, v5)), Var(v5)), Var(v6)))),
               Exists(v7, App(App(App(Var(v2), Store(0, v5)), Var(v6)),
               Var(v7))))))))))
        end
      )
    | tokenize "no" = T("no",article,
        let
            val v1 = get_new_var()
            val v2 = get_new_var()
            val v3 = get_new_var()
            val v4 = get_new_var()
            val v5 = get_new_var()
            val v6 = get_new_var()
            val v7 = get_new_var()
        in
            Abs(v1, Abs(v2, Abs(v3, Abs(v4, Conj(Eq(Var(v3), Var(v4)),
               Not(Exists(v5, Exists(v6, Exists(v7, Conj(Conj(Differs([],
               v3, v5), App(App(App(Var(v1), Store(0, v5)), Var(v5)),
               Var(v6))), App(App(App(Var(v2), Store(0, v5)), Var(v6)),
               Var(v7)))))))))))
        end
    )


  (*Conjunctions:*)
    | tokenize "and" = T("and", conjunction,
        let
            val v1 = get_new_var()
            val v2 = get_new_var()
            val v3 = get_new_var()
            val v4 = get_new_var()
            val v5 = get_new_var()
        in
            Abs(v1, Abs(v2, Abs(v3, Abs(v4, Exists(v5,
               Conj(App(App(Var(v1), Var(v3)), Var(v5)), App(App(Var(v2),
               Var(v5)), Var(v4))))))))
        end
    )
    | tokenize "or" = T("or", conjunction,
        let
            val v1 = get_new_var()
            val v2 = get_new_var()
            val v3 = get_new_var()
```

```
        val v4 = get_new_var()
        val v5 = get_new_var()
        val v6 = get_new_var()
    in
        Abs(v1, Abs(v2, Abs(v3, Abs(v4, Conj(Eq(Var(v3), Var(v4)),
           Disj(Exists(v5, App(App(Var(v1), Var(v3)), Var(v5))),
           Exists(v6, App(App(Var(v2), Var(v3)), Var(v6))))))))
    end
 )
 | tokenize "so" = T("so", conjunction,
    let
        val v1 = get_new_var()
        val v2 = get_new_var()
        val v3 = get_new_var()
        val v4 = get_new_var()
        val v5 = get_new_var()
        val v6 = get_new_var()
    in
        Abs(v1, Abs(v2, Abs(v3, Abs(v4, Conj(Eq(Var(v3), Var(v4)),
           Forall(v5, Disj(Not(App(App(Var(v1), Var(v3)), Var(v5))),
           Exists(v6, App(App(Var(v2), Var(v5)), Var(v6)))))))))
    end
  )

(*"Who" is a special case:*)
 | tokenize "who" = T("who", (N \ N ) / (VP),
    let
        val v1 = get_new_var()
        val v2 = get_new_var()
        val v3 = get_new_var()
        val v4 = get_new_var()
        val v5 = get_new_var()
        val v6 = get_new_var()
    in
        Abs(v1, Abs(v2, Abs(v3, Abs(v4, Abs(v5, Exists(v6,
           Conj(App(App(App(Var(v2), Var(v3)), Var(v4)), Var(v6)),
           App(App(App(Var(v1), Var(v3)), Var(v6)), Var(v5))))))))
    end
  )


(*"Is" is a special case:*)
 | tokenize "is" = T("is", verb_monotransitive,
    let
        val v1 = get_new_var()
        val v2 = get_new_var()
        val v3 = get_new_var()
```

```
          val v4 = get_new_var()
          val v5 = get_new_var()
      in
          Abs(v1,Abs(v2,App(Var(v1),Abs(v3,Abs(v4, Abs(v5,
             Conj(Eq(Var(v3),Var(v2)), Eq(Var(v4), Var(v5))))))))))
      end
  )



(*Raise exception on unknown words:*)
  | tokenize _ = raise UnknownWord


(*The indexer needs a list of all proper nouns:*)
val proper_noun_list = ["alice", "bob", "carl", "denise", "alex"]
```

Appendix D

Test Suite

```
(*-----------------------------------------------------------------*)
(* Functions defining a testing environment. Thorough, albeit by *)
(* nature incomplete.                                           *)
(*-----------------------------------------------------------------*)

val examples =[
(*The detailed example used in the report:*)
("a farmer beats a donkey. the poor animal hates him",
"Ev1.Ev2.(((farmer v1) & ((donkey v2) & ((beats v2) v1))) & (((animal
   v2) & (poor v2)) & ((hates v1) v2)))"),


(*The examples used by Muskens:*)
("a farmer owns a donkey. the bastard beats it",
"Ev1.Ev2.(((farmer v1) & ((donkey v2) & ((owns v2) v1))) & ((bastard
   v1) & ((beats v2) v1)))"),

("every farmer who owns a donkey beats it",
"Av1.Av2.((~(farmer v1) | (~(donkey v2) | ~((owns v2) v1))) | ((beats
   v2) v1))"),


(*Simple cases:*)
("a man sleeps",
"Ev1.((man v1) & (sleeps v1))"),
```

```
("a man loves a woman",
"Ev1.Ev2.((man v1) & ((woman v2) & ((loves v2) v1)))"),

("every man sleeps",
"Av1.(~(man v1) | (sleeps v1))"),

("no man sleeps",
"Av1.(~(man v1) | ~(sleeps v1))"),

("every man loves a woman",
"Av1.(~(man v1) | Ev2.((woman v2) & ((loves v2) v1)))"),

("no man loves every woman",
"Av1.(~(man v1) | Ev2.((woman v2) & ~((loves v2) v1)))"),

(*Examples with 'who':*)
("carl hates a farmer who owns a donkey",
"Ev1.Ev2.(((farmer v1) & ((donkey v2) & ((owns v2) v1))) & ((hates v1)
   carl))"),

("a farmer who owns a donkey hates every bastard who beats the donkey",
"Ev1.Ev2.(((farmer v1) & ((donkey v2) & ((owns v2) v1))) &
   Av3.((~(bastard v3) | (~(donkey v2) | ~((beats v2) v3))) | ((hates
   v3) v1)))"),

("every farmer who owns a donkey hates every bastard who beats the
   donkey",
"Av1.Av2.((~(farmer v1) | (~(donkey v2) | ~((owns v2) v1))) |
   Av3.((~(bastard v3) | (~(donkey v2) | ~((beats v2) v3))) | ((hates
   v3) v1)))"),


(*Examples with adjectives:*)
("a good man loves a bad woman",
"Ev1.Ev2.(((man v1) & (good v1)) & (((woman v2) & (bad v2)) & ((loves
   v2) v1)))"),

("a bad farmer who owns a lazy donkey beats the poor animal.",
"Ev1.Ev2.((((farmer v1) & (bad v1)) & (((donkey v2) & (lazy v2)) &
   ((owns v2) v1))) & (((animal v2) & (poor v2)) & ((beats v2) v1)))"),


(*Examples with conjunctions:*)
("a woman sleeps and a man sleeps",
"Ev1.Ev2.(((woman v1) & (sleeps v1)) & ((man v2) & (sleeps v2)))"),
```

```
("every woman sleeps and every man sleeps",
"(Av1.(~(woman v1) | (sleeps v1)) & Av2.(~(man v2) | (sleeps v2)))"),

("no farmer beats a donkey or every farmer beats a donkey",
"(Av1.Av2.(~(farmer v1) | (~(donkey v2) | ~((beats v2) v1))) |
   Av3.(~(farmer v3) | Ev4.((donkey v4) & ((beats v4) v3))))"),

("a farmer owns a donkey so he beats it",
"Av1.Av2.((~(farmer v1) | (~(donkey v2) | ~((owns v2) v1))) | ((beats
   v2) v1))"),

("a farmer beats a donkey so the donkey hates the farmer",
"Av1.Av2.((~(farmer v1) | (~(donkey v2) | ~((beats v2) v1))) | ((donkey
   v2) & ((farmer v1) & ((hates v1) v2))))"),

("a farmer owns a lazy donkey so the bastard beats it",
"Av1.Av2.((~(farmer v1) | ((~(donkey v2) | ~(lazy v2)) | ~((owns v2)
   v1))) | ((bastard v1) & ((beats v2) v1)))"),

("a man loves a woman and the woman loves the man",
"Ev1.Ev2.(((man v1) & ((woman v2) & ((loves v2) v1))) & ((woman v2) &
   ((man v1) & ((loves v1) v2))))"),

("a man sleeps or every man sleeps",
"(Ev1.((man v1) & (sleeps v1)) | Av2.(~(man v2) | (sleeps v2)))"),

("a man sleeps or no man sleeps",
"(Ev1.((man v1) & (sleeps v1)) | Av2.(~(man v2) | ~(sleeps v2)))"),

("a man who loves a woman sleeps or a woman who loves a man sleeps",
"(Ev1.Ev2.(((man v1) & ((woman v2) & ((loves v2) v1))) & (sleeps v1)) |
   Ev3.Ev4.(((woman v3) & ((man v4) & ((loves v4) v3))) & (sleeps
   v3)))"),


(*Examples with proper nouns:*)
("alice loves bob.",
"((loves bob) alice)"),

("alice loves bob and bob loves alice",
"(((loves bob) alice) & ((loves alice) bob))"),

("carl hates bob. bob owns a donkey.",
"Ev1.(((hates bob) carl) & ((donkey v1) & ((owns v1) bob)))"),

("carl hates bob. carl owns a donkey.",
"Ev1.(((hates bob) carl) & ((donkey v1) & ((owns v1) carl)))"),
```

```
("alex loves a man. she sleeps",
"Ev1.(((man v1) & ((loves v1) alex)) & (sleeps alex))"),

("alex loves a woman. he sleeps",
"Ev1.(((woman v1) & ((loves v1) alex)) & (sleeps alex))"),

("a man who loves alice owns a donkey so alice loves the donkey",
"Av1.Av2.(((~(man v1) | ~((loves alice) v1)) | (~(donkey v2) | ~((owns
   v2) v1))) | ((donkey v2) & ((loves v2) alice)))"),

("alice loves every farmer who beats no donkey so alice loves bob",
"(Ev1.(((farmer v1) & Av2.(~(donkey v2) | ~((beats v2) v1))) & ~((loves
   v1) alice)) | ((loves bob) alice))"),


(*Examples with 'is':*)
("bob is a farmer",
"(farmer bob)"),

("alice loves bob. he is a good man",
"(((loves bob) alice) & ((man bob) & (good bob)))"),

("every man who is a good farmer owns a donkey",
"Av1.((~(man v1) | (~(farmer v1) | ~(good v1))) | Ev2.((donkey v2) &
   ((owns v2) v1)))"),

("every farmer who beats no donkey is a good farmer",
"Av1.((~(farmer v1) | Ev2.((donkey v2) & ((beats v2) v1))) | ((farmer
   v1) & (good v1)))"),

("no man who is a good farmer beats a donkey",
"Av1.Av2.((~(man v1) | (~(farmer v1) | ~(good v1))) | (~(donkey v2) |
   ~((beats v2) v1)))"),

("no good farmer beats every donkey so bob is a bad farmer",
"(Ev1.(((farmer v1) & (good v1)) & Av2.(~(donkey v2) | ((beats v2)
   v1))) | ((farmer bob) & (bad bob)))"),


(*Complicated composite cases:*)
("a farmer owns a donkey and it is a lazy bastard so the farmer beats
   the donkey",
"Av1.Av2.(((~(farmer v1) | (~(donkey v2) | ~((owns v2) v1))) |
   (~(bastard v2) | ~(lazy v2))) | ((farmer v1) & ((donkey v2) &
   ((beats v2) v1))))"),
```

```
("bob beats no donkey so he is a good farmer",
"(Ev1.((donkey v1) & ((beats v1) bob)) | ((farmer bob) & (good bob)))"),

("no man who is a good farmer loves a woman who beats a donkey",
"Av1.Av2.Av3.((~(man v1) | (~(farmer v1) | ~(good v1))) | ((~(woman v2)
    | (~(donkey v3) | ~((beats v3) v2))) | ~((loves v2) v1)))")
];


(*Try a specific test:*)
fun try_example (text, expected) = (translate text) = expected;

(*Reference to the test on which full_test failed:*)
val failed_on = ref "none";

(*Try a list of tests:*)
fun test ((h1, h2)::t) =
      let
         val void = failed_on := h1
      in
         if not((try_example (h1, h2))) then false
         else (test t)
      end
  | test [] = true;

(*Wrapper function:*)
fun full_test () = test examples;
```

Bibliography

[All95] J. Allen. Natural language understanding. Benjamin/Cummings series in computer science. Benjamin/Cummings Pub. Co., 2nd edition, 1995.

[Bar92] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. M. Gabbay, and T. S. E. Maibaum, editors, Handbook of Logic in Computer Science, volume 2, pages 117–309. Oxford University Press, 1992.

[Cho93] N. Chomsky. Lectures on Government and Binding: The Pisa Lectures. Studies in generative grammar / Studies in generative grammar. Gruyter, Walter de GmbH, 1993.

[GHC98] N. Ge, J. Hale, and E. Charniak. A statistical approach to anaphora resolution. In Proceedings of the Sixth Workshop on Very Large Corpora, pages 161–170, 1998.

[HKT84] D. Harel, D. Kozen, and J. Tiuryn. Dynamic logic. In Handbook of Philosophical Logic, pages 497–604. MIT Press, 1984.

[Jac02] R. Jackendoff. Foundations of Language:Brain, Meaning, Grammar, Evolution. OUP Oxford, 2002.

[JM09] D.S. Jurafsky and J.H. Martin. Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition. Pearson international edition. Pearson Prentice Hall/Pearson education international, 2009.

[Mat12] D. Matthews. Poly/ml. `http://www.polyml.org/`, 2012.

[Mon70] R. Montague. Universal grammar. Theoria, 36(3):373–398, 1970.

[MR12] R. Moot and C. Retoré. The Logic of Categorial Grammars: A Deductive Account of Natural Language Syntax and Semantics. FoLLI publications on logic, language and information. Springer-Verlag Berlin Heidelberg, 2012.

[Mus91] R. Muskens. Anaphora and the logic of change. In J. van Eyck, editor, Logics in AI, volume 478 of Lecture Notes in Artificial Intelligence, pages 412–428. Springer, 1991.

[PNW13] L. Paulson, T. Nipkow, and M. Wenzel. Isabelle proof assistant. `http://www.cl.cam.ac.uk/research/hvg/Isabelle/`, 2013.

[Ran94] A. Ranta. Type-Theoretical Grammar. Indices (Claredon). Clarendon Press, 1994.

[Sau16] F. Saussure. Course in General Linguistics. McGraw-Hill Book Company, 1916.