An Inverse Lambda Calculus Algorithm

For Natural Language Processing

by

Marcos Alvarez Gonzalez

A Thesis Presented In Partial Fulfillment
of the Requirements for the Degree
Master of Science

Approved November 2011 by the
Graduate Supervisory Committee:

Chitta Baral, Chair
Joohyung Lee
Jieping Ye

ARIZONA STATE UNIVERSITY

November 2010
ABSTRACT

Natural Language Processing is a subject that combines computer science and linguistics, aiming to provide computers with the ability to understand natural language and to develop a more intuitive human-computer interaction. The research community has developed ways to translate natural language to mathematical formalisms. It has not yet been shown, however, how to automatically translate different kinds of knowledge in English to distinct formal languages. Most of the recent work presents the problem that the translation method aims to a specific formal language or is hard to generalize.

In this research, I take a first step to overcome this difficulty and present two algorithms which take as input two lambda-calculus expressions G and H and compute a lambda-calculus expression F. The expression F returned by the first algorithm satisfies F@G=H and, in the case of the second algorithm, we obtain G@F=H. The lambda expressions represent the meanings of words and sentences. For each formal language that one desires to use with the algorithms, the language must be defined in terms of lambda calculus. Also, some additional concepts must be included. After doing this, given a sentence, its representation and knowing the representation of several words in the sentence, the algorithms can be used to obtain the representation of the other words in that sentence. In this work, I define two languages and show examples of their use with the algorithms.

The algorithms are illustrated along with soundness and completeness proofs, the latter with respect to typed lambda-calculus formulas up to the second order. These algorithms are a core part of a natural language semantics system that translates sentences from English to formulas in different formal languages.
ACKNOWLEDGMENTS

I am grateful to everyone that made this work possible. Especially to Dr. Baral for his review, help and comments, Aaron Gottesman and Juraj Dzifcak for their useful insights.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF TABLES</td>
<td>v</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>vi</td>
</tr>
<tr>
<td>INTRODUCTION</td>
<td>1</td>
</tr>
<tr>
<td>Main contributions</td>
<td>2</td>
</tr>
<tr>
<td>Thesis Outline</td>
<td>3</td>
</tr>
<tr>
<td>TYPED LAMBDA CALCULUS</td>
<td>4</td>
</tr>
<tr>
<td>Typed First-Order Logic Lambda Calculus</td>
<td>13</td>
</tr>
<tr>
<td>Typed Answer Set Programming Lambda Calculus</td>
<td>26</td>
</tr>
<tr>
<td>Type Order</td>
<td>38</td>
</tr>
<tr>
<td>The Inverse Lambda Algorithms</td>
<td>46</td>
</tr>
<tr>
<td>INVERSE LAMBDA CALCULUS ALGORITHMS</td>
<td>49</td>
</tr>
<tr>
<td>The inverse lambda operators</td>
<td>49</td>
</tr>
<tr>
<td>Deterministic version of the Inverse Lambda Algorithms</td>
<td>52</td>
</tr>
<tr>
<td>Definition of the Inverse Lambda Algorithm and Complexity Analysis</td>
<td>54</td>
</tr>
<tr>
<td>Complexity Analysis and Implementation</td>
<td>58</td>
</tr>
<tr>
<td>EXAMPLES AND USE CASES</td>
<td>65</td>
</tr>
<tr>
<td>Inverse Lambda Algorithm Examples</td>
<td>65</td>
</tr>
<tr>
<td>Example 1 - Typed FOL lambda calculus</td>
<td>65</td>
</tr>
<tr>
<td>Example 2 - Typed FOL lambda calculus</td>
<td>65</td>
</tr>
<tr>
<td>Example 3 - Typed FOL lambda calculus</td>
<td>66</td>
</tr>
<tr>
<td>Example 4 - Typed ASP lambda calculus</td>
<td>67</td>
</tr>
<tr>
<td>Example 5 - Typed ASP lambda calculus</td>
<td>67</td>
</tr>
<tr>
<td>Example 6 - Typed FOL lambda calculus</td>
<td>68</td>
</tr>
<tr>
<td>Example 7 - Typed FOL lambda calculus</td>
<td>68</td>
</tr>
<tr>
<td>Example 8 - Typed ASP lambda calculus</td>
<td>70</td>
</tr>
<tr>
<td>Example 9 - Typed ASP lambda calculus</td>
<td>70</td>
</tr>
<tr>
<td>Example 10 - Typed FOL lambda calculus</td>
<td>71</td>
</tr>
<tr>
<td>Example 11 - Typed FOL lambda calculus</td>
<td>72</td>
</tr>
<tr>
<td>Use Cases</td>
<td>72</td>
</tr>
<tr>
<td>Background: CCG parsing</td>
<td>73</td>
</tr>
<tr>
<td>Increasing the lexicon of sentences in Typed FOL lambda calculus</td>
<td>76</td>
</tr>
<tr>
<td>Increasing the lexicon of normative sentences and exceptions in Typed ASP lambda calculus</td>
<td>78</td>
</tr>
<tr>
<td>THEOREMS AND PROOFS</td>
<td>81</td>
</tr>
<tr>
<td>CLOSE RELATED WORK</td>
<td>100</td>
</tr>
<tr>
<td>Higher-order matching and Interpolation</td>
<td>100</td>
</tr>
<tr>
<td>Learning word-to-meaning mappings</td>
<td>104</td>
</tr>
<tr>
<td>CONCLUSION</td>
<td>108</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>111</td>
</tr>
</tbody>
</table>
## LIST OF TABLES

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Examples of typed FOL lambda calculus formulas and their types.</td>
<td>17</td>
</tr>
<tr>
<td>2. Possible order combinations for F, G and H formulas.</td>
<td>46</td>
</tr>
<tr>
<td>3. CCG parsing for “Joe eats pasta”.</td>
<td>74</td>
</tr>
<tr>
<td>4. semantic tree for “Joe eats pasta”.</td>
<td>76</td>
</tr>
<tr>
<td>5. CCG and λ-calculus derivation for “Texas borders a state, what state borders Texas?”</td>
<td>76</td>
</tr>
<tr>
<td>6. CCG and λ-calculus derivation for “Most birds fly” and “Penguins are birds”.</td>
<td>79</td>
</tr>
<tr>
<td>7. CCG and λ-calculus derivation for “Penguins do swim” and “Penguins do not fly”.</td>
<td>80</td>
</tr>
</tbody>
</table>
# LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.</td>
<td>Annotated derivation tree for the first typed FOL $\lambda$-calculus example formula.</td>
<td>21</td>
</tr>
<tr>
<td>2.</td>
<td>Annotated derivation tree for the fourth typed FOL $\lambda$-calculus example formula.</td>
<td>22</td>
</tr>
<tr>
<td>3.</td>
<td>Annotated derivation tree for the first typed FOL $\lambda$-calculus example that is not a formula.</td>
<td>23</td>
</tr>
<tr>
<td>4.</td>
<td>Annotated derivation tree for the fourth typed ASP $\lambda$-calculus example.</td>
<td>35</td>
</tr>
<tr>
<td>5.</td>
<td>Annotated derivation tree for the second typed ASP $\lambda$-calculus example that is not a formula.</td>
<td>36</td>
</tr>
<tr>
<td>6.</td>
<td>Binary tree representation for a third order type where the depth of the most left-nested arrow is three.</td>
<td>43</td>
</tr>
<tr>
<td>7.</td>
<td>Binary tree representation for a second order type where the depth of the most left-nested arrow is two.</td>
<td>43</td>
</tr>
</tbody>
</table>
1. INTRODUCTION

The overall goal of this proposed research is to translate English to formal logics and Knowledge Representation (KR) languages. In order to reach this objective, the approach consists of using \( \lambda \)-calculus formulas [1] to represent the meaning of words and sentences as previously shown in [2–5]. Combinatorial Categorical Grammar (CCG) [6] is used to construct the meaning of a sentence from the meaning of its constituent words and phrases. The above technique incorporates the complex challenge of obtaining the \( \lambda \)-calculus formulas that represent the meaning of each of the words. Some of the recent approaches [3–5] use a handcrafted set of formulas or a limited set of rules while focusing on a particular representation formalism to learn new semantic information. The approach that will be presented in this work is independent of the selected formalism and does not use any predefined rules or formulas. By analyzing how a human would derive the \( \lambda \)-calculus formulas of words, it can be observed that, at times, a human given \( G \) and \( H \), would try to construct a formula \( F \) such that \( F@G=H \) or \( G@F=H \).

In this work, this idea is formalized and presented with two algorithms that compute \( F \) given \( G \) and \( H \); in the case of the first algorithm \( F@G=H \) and in the case of the second algorithm \( G@F=H \). These algorithms are referred to as the Inverse \( \lambda \)-Algorithms. In the absence of a single KR language that is appropriate for expressing all the nuances of a natural language, the algorithms are defined so that they can be useful with respect to multiple KR languages. In this work, its use is described with First-Order Logic and Answer Set Programming. For each formalism, a new language is defined, which merges them with the theory of Typed Lambda Calculus. The signature, syntax and semantics of the new languages are defined, based on the theory from the original formalisms.
This set of languages is present in the works [3, 4] which follow a similar approach to translate text into a particular appropriate knowledge representation language. In [3], they use lambda-calculus expressions and first-order logic to map sentences to their logical form from a geographic database and job listings database. In [4], Answer Set Programming and lambda-calculus are utilized to represent sentences with default and normative statements. In a similar work, [5], the authors take English words where their semantics are represented in temporal and dynamic logic extended by λ-calculus, and they map them to temporal and dynamic logic formulas which express goals and actions of a robot.

Soundness and completeness results of the algorithms are presented. The completeness result is with respect to typed lambda-calculus formulas up to the second order. In a companion work, a learning-based natural language semantics system that uses the algorithms was developed and compared to other similar systems in the literature. A matter of current research is to enhance the algorithms to be complete when considering typed lambda-calculus formulas up to the third order.

A comparison with close related work will also be discussed. Two other similar problems found in related literature will be presented. The similarities and differences with the research introduced in this work are considered and some new terminology is defined. The possible idea of considering the problem presented in this research, as a special case of the related problems, is left for future work and study.

1.1. Main contributions

In this research, we introduce Two Inverse λ-Algorithms and their application to typed First-Order Logic λ-calculus and Typed Answer Set Programming λ-calculus. The two languages are defined in terms of typed lambda calculus and it is shown how derivation
trees and type orders are defined for two formalisms.

The main contribution of this work is that it makes possible to automatically obtain semantic representations of unknown words using the information already available from known sentences and words. Also, the Inverse $\lambda$-Algorithms can be used with any formalism that one may desire, it just needs to be defined in terms of typed lambda calculus. This is an important difference with respect to previous approaches.

The Inverse $\lambda$-Algorithms are an important part of a natural language learning system which learns the semantic representations of words from sentences. The use of the algorithms allow the system to take advantage of this novel approach and the results obtained by the overall system have shown better results than previous approaches for several standard corpuses.

1.2. Thesis Outline

The rest of this work is organized as follows: chapter 2 presents background material on typed lambda calculus, and the definition and study of each of the formal languages considered in this work. Chapter 3 presents the Inverse $\lambda$-Algorithms. Chapter 4 illustrates the algorithms with respect to several examples and shows use case examples with the different languages. Chapter 5 presents the soundness and completeness results. Chapter 6 discusses close related work to the research presented in this thesis and Chapter 7 finishes this work with a conclusion.
2. TYPED LAMBDA CALCULUS

In the field of linguistics and natural language semantics, $\lambda$-calculus has been recognized as an adequate and concise way of representing English syntactic constructions. This is because most of these constructions, such as the English construction where a sentence can consist of a noun phrase and a verb phrase, can be interpreted as a function and argument application. This behavior coincides with how $\lambda$-abstraction and application works in $\lambda$-calculus. Montague, [7], was the first to suggest that natural languages need not be treated differently from formal languages and that they could have formal semantics. The approach of using $\lambda$-calculus to represent the meaning of words and $\lambda$-application as a mechanism to construct the meaning of phrases and sentences is also considered to originate from Montague. As aforementioned, that is also the approach in this work.

However, to further ground the notion of “meaning” (or semantics) it is useful to have a notion of models of $\lambda$-calculus expressions. Such notions also allow evaluation of natural language meaning representations. When referring to model, one is looking for a semantic tool that can give me two elements: the entities that are part of the domain, and for every element in the signature, the semantic value associated with it. Following the theory from [2], consider a first-order logic model consisting of a domain $D$, which states the entities that are going to be considered, and an interpretation function $F$, which denotes semantic values in $D$. Therefore, a model $M$ consists of a pair $(D,F)$. Now, setting a model for a typed $\lambda$-calculus system based on first-order logic is rather intuitive. One just takes a first-order model $M$ and assigns denotations of types to elements of the domain, expressions that can be true or false and expressions that are functions from one
type to another. These concepts will be explained in detail in section 2.1 of this work. By creating this model with the corresponding denotations for types, expressions of the system will have a defined type and semantic value associated.

Both untyped and typed $\lambda$-calculus can be characterized using models, but the one that has had the most impact on natural language semantics is typed $\lambda$-calculus, as creating models for typed $\lambda$-calculus theories is relatively simple. It also became familiar to linguistics after the mentioned Montague’s works.

“Simply Typed Lambda Calculus” of Church [8] will be used, since it is the most commonly used in linguistics. In this theory, only one type constructor is considered to build function types: “$\rightarrow$” and each term has a single type associated with it [9].

Next, some $\lambda$-calculus definitions from [10] and [11] are presented as initial background knowledge for this work. The signature of the Lambda Calculus is presented first:

**Definition 1** [Lambda Calculus signature]

- The lambda operator $\lambda$.
- The lambda application $\@$.
- The parenthesis (,).
- An infinite set of variables $v_n$ for each natural number $n$.
- A (possibly empty) set of constants $c$.

**Definition 2** [untyped term] Given an infinite set of variables and a set of constants from the signature. The set of terms is defined inductively as follows:

- All variables and constants are terms. (called atomic terms)
• If M and N are terms, then (M@N) is a term. (called application)

• If M is a term and x is a variable, then (λx.M) is a term. (called abstraction)

• Nothing else is a term.

**Example 1:** Examples of terms are:

- λx.(x@y).

- (λy.y)@((λx.(x@y))).

- (λx.x)@y

Examples that are not terms are:

- λx.@y.

- (λy)@((λx.(x@y))).

- (λ.x)@y

**Definition 3** [Occurrence] Let P and Q be λ-calculus terms, the relation Q has an occurrence of P is defined as:

1. P occurs in P

2. if P occurs in M or in N, then P occurs in (M@N).

3. if P occurs in M or P ≡ x, then P occurs in (λx.M).

**Example 2:** Examples of occurrences are:

- The term y occurs in λx.(x@y).
• The term \((\lambda x.(x@y))\) occurs in \((\lambda y.y)@((\lambda x.(x@y)))\).

• The term \(x\) occurs in \((\lambda x.x)@y\).

**Definition 4** [sub-term] A sub-term of a \(\lambda\)-calculus term \(F\) is any term \(P\) that occurs in \(F\).

**Example 3:** Examples of sub-term are:

- A sub-term of \(\lambda x.(x@y)\) is \((x@y)\).
- A sub-term of \((\lambda y.y)@((\lambda x.(x@y)))\) is \((\lambda y.y)\).
- A sub-term of \((\lambda x.x)@y\) is \(y\).

Examples that are not sub-terms are:

- \(\lambda x\) is not a sub-term of \(\lambda x.(x@y)\).
- \((\lambda x.x)\) is not a sub-term of \((\lambda y.y)@((\lambda x.(x@y)))\).

**Definition 5** [scope] Let \(\lambda v.P\) be a term. The scope of the abstractor \(\lambda v\) is \(P\).

**Example 4:** In the term \((\lambda y.y)@((\lambda x.(x@y)))\), the variable \(y\) in the first term of the application is in the scope of \(\lambda y\). The term \((x@y)\) is in the scope of \(\lambda x\) in the second term of the application. In the term \((\lambda x.((\lambda y.(x@y))))\), the scope of \(\lambda x\) is \((\lambda y.(x@y)))\).

**Definition 6** [Free and bound variables] An occurrence of a variable \(x\) in a term \(P\) is bound if and only if it is in the scope of an occurrence of \(\lambda x\) in \(P\), otherwise it is free. If \(x\) has at least one free occurrence in \(P\), it is called a free variable of \(P\); the set of all such variables is denoted by \(FV(P)\).
Example 5: In the term $\lambda x. (xy)$, the variables $x$ is bound and $y$ is free. In the term $(\lambda y.y)@\lambda x. (xy)$, the variable $y$ has a bound occurrence in the first term of the application and a free occurrence in the second term.

Definition 7 [Substitution] Define $[N/x]M$ to be the result of substituting $N$ for each free occurrence of $x$ in $M$, and changing bound variables to avoid variable clashing. A more formal definition is the following:

\[
\begin{align*}
(a) & \quad [N/x]x \equiv N \\
(b) & \quad [N/x]a \equiv a \quad \forall \text{atom } a \neq x \\
(c) & \quad [N/x](PQ) \equiv ([N/x]P[N/x]Q) \\
(d) & \quad [N/x](\lambda x.P) \equiv \lambda x.P \\
(e) & \quad [N/x](\lambda y.P) \equiv \lambda y.([N/x]P) y \neq x \land y \not\in \text{FV}(N) \lor x \not\in \text{FV}(P) \\
(f) & \quad [N/x](\lambda y.P) \equiv \lambda z.([N/x][z/y]P) y \neq x \land y \not\in \text{FV}(N) \land d \in \text{FV}(P)
\end{align*}
\]

In (f), $z$ is chosen to be the first variable $\not\in \text{FV}(NP)$.

Example 6: Examples of substitutions are:

- The $\beta$-redex $(\lambda x.x)@y$ corresponds to the substitution $[y/x]x$ and one applies rule (a) above.

- The $\beta$-redex $(\lambda x.(x@y))@M$ corresponds to the substitution $[M/x](x@y)$ and one applies rule (c).
• The $\beta$-redex $(\lambda x.((x@y))@M$ corresponds to the substitution $[M/x](\lambda x.(x@y))$ and one applies rule (d).

• The $\beta$-redex $(\lambda x.(\lambda y.(y@x)))(\lambda z.z)$ corresponds to the substitution $[M/x](\lambda y.(y@x))$ where $M = (\lambda z.z)$ and one applies rule (e).

• The $\beta$-redex $(\lambda x.\lambda y.(x@y))@y$ corresponds to the substitution $\lambda z.[y/x][z/y](x@y)$ and one applies rule (f).

**Definition 8** [β-redex] A $\beta$-redex is a term of the form $(\lambda x.M)@N$ where $M$ and $N$ are terms. The term $[N/x]M$ is called its contractum.

**Example 7:** An example of a $\beta$-redex is the term $(\lambda x.x)@y$. The term $(x)@y$ is not a $\beta$-redex.

**Definition 9** [β-contraction] If a $\lambda$-calculus term $P$ contains a $\beta$-redex occurrence $R$ and one replaces that occurrence by $[N/x]M$, and the result is $Q$, then it is said that $P$ $\beta$-contracts to $Q$.

**Example 8:** Consider $P$ a term with a $\beta$-redex. For example, $(\lambda x.x)@y$. Consider $Q$ the term $y$, result of the substitution $[y/x]x$ from the application of the $\beta$-redex of $P$. In this case, $(\lambda x.x)@y$ $\beta$-contracts to $y$.

To properly understand the behavior of the application of terms $(M@N)$, it is necessary to have a clear idea of the role that parentheses play. The location of parentheses inside applications or separating terms denotes the order and result. The following example illustrates an important difference:
Example 9: Consider the following two terms which differ only in the position of the parentheses:

1. \( \lambda x. (x@y) \).

2. \( (\lambda x.x)@y \).

The first term will become a \( \beta \)-redex \((\lambda x.M)@N\) when a term \( N \) is applied to \((\lambda x.M)\) where \( M = x@y \). The second term is already a \( \beta \)-redex and after the \( \beta \)-contraction one obtains the term \( y \). In the first case, if we apply to the expression the term \( N \), one has \((\lambda x.(x@y))@N\) which \( \beta \)-contracts to \( N@y \). Now the term \( y \) is applied to the term \( N \) and the necessary substitutions will be performed in the term \( N \). Thus, one can see how two expression that look like the same term, actually have different behavior because of the parentheses.

Definition 10 \([\alpha\text{-conversion}]\) Let \( M \) be a term. Let \( y \) be a variable that does not occur in \( M \). The act of replacing an occurrence of \( \lambda x.M \) in a term by \( \lambda y. [y/x]M \) is called an \( \alpha \)-conversion. If \( P \) changes to \( Q \) by a finite series of \( \alpha \)-conversions, we say that \( P \) \( \alpha \)-converts to \( Q \).

Example 10: Examples of \( \alpha \)-conversions are:

- The term \( \lambda x.(x@y) \) \( \alpha \)-converts to \( \lambda z.(z@y) \).

- The term \( (\lambda y.y)@(\lambda x.(x@y)) \) \( \alpha \)-converts to \( (\lambda w.w)@(\lambda v.(v@y)) \).

The \( \alpha \)-conversion plays an important role when we have variable clashes during substitutions. Consider the following \( \beta \)-redex:

- \( (\lambda x.\lambda y.(x@y))@y \).
In this case, one needs to perform an \( \alpha \)-conversion before the application. By doing this, the variable \( y \) does not get bound to the lambda abstracter \( \lambda y \). After the \( \alpha \)-conversion \( \lambda z.\left[ z/y \right] M \) where \( M = (\lambda x.\lambda z.(x@z)) \), the \( \beta \)-redex now is as follows:

- \((\lambda x.\lambda z.(x@z))@y\).

**Definition 11** [closed term] A closed term is a term in which no variable occurs free.

**Example 11:** Examples of closed terms are:

- \( \lambda x.\lambda y.(x@y) \).
- \( (\lambda y.y)@ (\lambda x.(x@y)) \).

Examples that are not closed terms are:

- \( \lambda x.(x@y) \).
- \( (\lambda y.y)@ (\lambda x.(x@z)) \).

**Definition 12** [\( \lambda I \)-term] A term \( P \) is called \( \lambda I \)-term if and only if, for each sub-term with form \( \lambda x. M \) in \( P \), \( x \) occurs free in \( M \) at least once.

**Example 12:** Examples of \( \lambda I \)-terms are:

- \( \lambda x.\lambda y.(x@y) \).
- \( (\lambda y.y)@ (\lambda x.(x@y)) \).

Examples that are not \( \lambda I \)-terms are:

- \( \lambda x.\lambda z.(x@y) \).
Definition 13 [β-normal form] A λ-calculus formula is in β-normal form if it does not contain any β-redex occurrences.

Example 13: Consider the term \(((\lambda x.(\lambda y.(x@y)))(\lambda x.x))@y\). Its β-normal form is \(y\). The β-contractions are:

- \((\lambda y.(\lambda x.x@y))@y\)
- \((\lambda y.y)@y\)
- \(y\)

This section ends with the introduction of simply typed lambda calculus syntax. As it is stated in [10], the definitions presented above are valid both for untyped and typed λ-calculus. Each of the languages defined in the following sections will present its own version of the following definitions of typed lambda calculus.

Definition 14 [Types] Assume that we have been given some symbols called base types; then we can define types as follows:

- Each base type is a type.
- If \(a\) and \(b\) are types, then \((a \rightarrow b)\) is a type.

Definition 15 [Typed Lambda Calculus signature]

- The lambda operator \(\lambda\)
- The lambda application @
• The parenthesis (,)

• For every type \( a \), an infinite set of variables \( v_{n,a} \) for each natural number \( n \)

• For every type \( a \), a (possibly empty) set of constants \( c_a \) of type \( a \)

**Definition 16** [Typed term] For each type \( a \), given an infinite set of variables of type \( a \) and a set of constants of type \( a \) from the signature. The set of typed terms is defined as follows:

- All variables and constants of type \( a \) are typed terms of type \( a \).

- If \( M_{a \rightarrow b} \) and \( N_a \) are typed terms of types \( (a \rightarrow b) \) and \( a \) respectively, then \( (M_{a \rightarrow b}@N_a) \) is a typed term of type \( b \).

- If \( M_b \) is a typed term of type \( b \) and \( x_a \) is a variable of type \( a \), then \( (\lambda x_a.M_b) \) is a typed term of type \( (a \rightarrow b) \).

- Nothing else is a typed term.

### 2.1. Typed First-Order Logic Lambda Calculus

First-Order Logic has been widely studied and used for many years. Since a large body of research in natural language semantics has focused on translating natural language to first-order logic, the use of the Inverse \( \lambda \)-Algorithms and first-order logic will be presented; particularly with Typed First-Order Logic Lambda Calculus. For obtaining formal results, the signature of the language, the construction of typed terms, typed formulas and the notion of occurrence of a typed term in the language, need to be defined.

This section is begun by introducing the signature of the Typed First-Order Logic Lambda Calculus Language (Typed FOL Lambda Calculus):
• The lambda operator \( \lambda \)

• The lambda application @

• The parenthesis \((, )\), \([, ]\)

• For each type \(a\), an infinite set of variables \(v_{n,a}\) for each natural number \(n\)

• For each type \(a\), a (possibly empty) set of constants \(c_a\) of type \(a\)

• The connectives \(\neg, \lor, \land, \rightarrow\)

• The quantifiers \(\forall\) and \(\exists\)

• Predicate symbols and function symbols with a given arity \(n\)

• The equality symbol =

**Definition 17** [FOL \(\lambda\)-element] Symbols of a typed first-order logic lambda calculus signature will be denoted as FOL \(\lambda\)-elements.

Next, the set of types that will be used with Typed FOL Lambda Calculus, in conjunction with the definition of the semantics of types assigned to the different expressions of the language will be introduced. The principles presented in [12] will be followed, where \(D_a\) represents the set of possible objects (denotations) that describe the meanings of expressions of type \(a\).

**Definition 18** The set of types \(\Theta\) is defined recursively as follows:

1. \(e\) is a type

2. \(t\) is a type
3. If \( a \) and \( b \) are types, then \((a \rightarrow b)\) is a type.

**Definition 19** Let \( E \) be a given domain of entities. Then the semantics are defined as:

- \( D_e = E \)
- \( D_t = \{0, 1\} \) the set of truth values
- \( D_{a \rightarrow b} = \) the set of functions from \( D_a \) to \( D_b \).

These letters are commonly used in natural language literature. An expression of type \( e \) denotes individuals that belong to the domain of a model. Expressions of type \( t \) denote truth values and they will be assigned to expressions that can be evaluated to a truth value in the model. Expressions of type \((a \rightarrow b)\) denote functions which input is in \( D_a \) and output values are in \( D_b \). For example, the type \( e \rightarrow t \) corresponds to functions from entities to truth values.

To continue, the definitions for FOL typed term and typed FOL \( \lambda \)-calculus formula are presented. The former is based on the structure of the typed terms definition presented in [12].

**Definition 20** A FOL atomic term is a constant or a variable of any type \( a \). If \( t_1, \ldots, t_n \) are FOL atomic terms, and \( f \) is a function symbol, \( f(t_1, \ldots, t_n) \) is also a FOL atomic term of type \( e \).

**Definition 21** The elements of the set \( \Delta_\alpha \) of FOL typed terms of type \( \alpha \) are inductively defined as follows:

1. For each type \( a \), every FOL atomic term of type \( a \) belong to \( \Delta_a \).
2. If $P$ is a predicate symbol of arity $n$ and $t_1, \ldots, t_n$ are FOL atomic terms then

$$P(t_1, \ldots, t_n)$$

is a FOL typed atomic formula that belongs to $\Delta_t$.

3. For any types $a$ and $b$, if $\alpha \in \Delta_{a \rightarrow b}$ and $\beta \in \Delta_a$, then $\alpha \@ \beta \in \Delta_b$.

4. For any types $a$ and $b$, if $u$ is a variable of type $a$ and $\alpha \in \Delta_b$ and has free occurrences of the variable $u$, then $\lambda u. \alpha \in \Delta_{a \rightarrow b}$ and the free occurrences of $u$ are now bound to the abstractor $\lambda u$.

5. If $\alpha \in \Delta_t$ and $\beta \in \Delta_t$, then $\alpha \lor \beta$, $\alpha \land \beta$ and $\alpha \rightarrow \beta \in \Delta_t$.

6. If $\alpha \in \Delta_t$, then $\neg \alpha \in \Delta_t$.

7. If $\alpha$ and $\beta \in \Delta_e$, then $(\alpha = \beta) \in \Delta_t$.

8. For any type $a$, if $\alpha \in \Delta_t$, and $u$ is a variable of type $a$, then $\forall u \alpha$ and $\exists u \alpha \in \Delta_t$.

**Definition 22** A typed FOL $\lambda$-calculus formula is a FOL typed term of type $a$ where every variable is bound to an abstractor (quantifier variables may not) and every abstractor binds to a variable.

These conditions ensure that first-order formulas are obtained when the typed FOL $\lambda$-calculus formula is in $\beta$-normal form. These two conditions correspond to closed and $\lambda$I-terms defined in the previous chapter. Next, some examples of formulas and their types are presented. The examples start from base types and continue generating more complex typed formulas. In this way, one can see the relation between types, formulas and their construction from the base types of the language.

---

1 Refer to the definition of occurrence presented at the end of this section.
<table>
<thead>
<tr>
<th>Formula</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>$e$</td>
</tr>
<tr>
<td>Mia</td>
<td>$e$</td>
</tr>
<tr>
<td>$\text{person}(\text{John})$</td>
<td>$t$</td>
</tr>
<tr>
<td>$\text{woman}(\text{Mia})$</td>
<td>$t$</td>
</tr>
<tr>
<td>$\lambda x. (\text{person}(x))$</td>
<td>$e \rightarrow t$</td>
</tr>
<tr>
<td>$\lambda x. \lambda y. (\text{person}(x) \land \text{woman}(y))$</td>
<td>$e \rightarrow (e \rightarrow t)$</td>
</tr>
<tr>
<td>$\lambda y. (\text{person}(\text{John}) \land \text{woman}(y))$</td>
<td>$e \rightarrow t$</td>
</tr>
<tr>
<td>$\lambda x. (\text{person}(\text{John}) \land \text{woman}(\text{Mia}))$</td>
<td>$t$</td>
</tr>
<tr>
<td>$\lambda x. (x@\text{John})$</td>
<td>$(e \rightarrow t) \rightarrow t$</td>
</tr>
<tr>
<td>$\exists x (\text{plane}(x) \land \text{takes}(\text{John}, x))$</td>
<td>$t$</td>
</tr>
</tbody>
</table>

Table 1

Examples of typed FOL lambda calculus formulas and their types.

Before presenting more examples of typed FOL $\lambda$-calculus formulas, an algorithm for obtaining the type of formulas is introduced. This algorithm can also be used to verify if a expression is well-typed and thus, if it can be built using the definition of typed terms. Using this algorithm one is able to deduce if a typed FOL $\lambda$-calculus expression is a typed term or not.

The algorithm presented will be based on derivation trees for context free grammars (CFG) [13] applied to lambda calculus expressions, known as abstract syntax trees [14]. The derivation trees that will be constructed follow the idea presented on [12], where each node has a type and derivation rule associated. The leaves of the tree represent variables and constants of a given type, while intermediate nodes correspond to FOL typed terms, built following the syntactic rules. For creating intermediate nodes, we will normally differentiate between unary and binary FOL $\lambda$-elements. A unary FOL $\lambda$-element will need one term to form another term, while binary FOL $\lambda$-elements will need two terms to form a new term. For example, a lambda abstractor $\lambda x$ will be a unary FOL
\( \lambda \)-element since it needs a term \( M \) to form a more complex term \( \lambda x. M \). The application symbol \( "@" \) will be a binary FOL \( \lambda \)-element, since we need two terms \( \alpha \) and \( \beta \) to form a new term \( \alpha @ \beta \). To construct the tree, typed terms with unary FOL \( \lambda \)-elements will have one child while binary FOL \( \lambda \)-elements will have two children. One can also have FOL \( \lambda \)-elements that will have more than two children like predicate symbols in the case of typed FOL lambda calculus. The arity of the predicate symbol will denote the number of children that it will have in the tree.

The algorithm starts creating the tree from the root, which corresponds to the typed term, and generates children as it reads the FOL \( \lambda \)-elements of the typed term. When the tree is built, we set the leaves with their corresponding types given in the input of the algorithm. Then, one starts going up using the derivation rules of the language labeling each node with the type generated by the rule and the rule number. The algorithm will be defined in a general way so that it can be used with the language that will be presented in the next section.

**Definition 23** [unary FOL \( \lambda \)-elements] The unary FOL \( \lambda \)-elements, identified as \( \upsilon \), correspond to the lambda operator \( \lambda \), the connective \( \neg \) and the quantifiers \( \forall \) and \( \exists \).

**Definition 24** [binary FOL \( \lambda \)-elements] The binary FOL \( \lambda \)-elements, identified as \( \omega \), correspond to the lambda application \( @ \), the connectives \( \land \), \( \lor \) and \( \rightarrow \) and the equality symbol \( = \).

**Definition 25** [n-ary FOL \( \lambda \)-elements] The n-ary FOL \( \lambda \)-elements, identified as \( \pi \), correspond to predicate symbols and function symbols of the signature.
Definition 26 [typed lambda calculus derivation tree algorithm] Consider $M_1, M_2, ..., M_n$ series of $\lambda$-elements.

- Input Algorithm Part 1:
  A series of $\lambda$-elements $M$.
  The types assigned to variables and constants in $M$.

- Output Algorithm Part 1:
  The derivation tree of $M$.

- Algorithm Part 1: $BuildTree(M)$
  1. If $M$ is a variable or constant of the signature, one adds a node with label $M$ and annotates the node with its type.
  2. If $M = vM_1$, one adds a node with label $M$ which child will have the label returned by $BuildTree(M_1)$.
  3. If $M = M_1 \omega M_2$, one adds a node with label $M$ which will have two children with the label returned by $BuildTree(M_1)$ and $BuildTree(M_2)$.
  4. If $M = \omega M_1 M_2 ... M_n$, one adds a node with label $M$ which will have $n$ children with the label returned by $BuildTree(M_1), BuildTree(M_2), ..., BuildTree(M_n)$.

- Input Algorithm Part 2:
  The derivation tree of $M$ from the Algorithm Part 1.

- Output Algorithm Part 2:
  The derivation tree of $M$ annotated with types and derivation rule numbers.
The type of \( M \) in the root of the tree or null if \( M \) is not a typed term.

- Algorithm Part 2: \textit{InferType}(\( ND \))

1. \( ND \) = leaves of the deepest level of the tree.

2. FOR \( i = 0 \) TO depth of tree-1 DO:

   - If nodes \( ND \) have labels \( M_1 \) and \( M_2 \) with types \( a \) and \( b \), and the parent node in the tree has the label \( M_1 \omega M_2 \), annotate the parent node with the corresponding number of the derivation rule of the language and type generated. If no derivation rule is found or types are null, exit.

   - Else If node \( ND \) has label \( M_1 \) and type \( a \), and the parent node in the tree has the label \( \upsilon M_1 \), annotate the parent node with the corresponding number of the derivation rule of the language and type generated. If no derivation rule is found or types are null, exit.

   - Else If nodes \( ND \) have labels \( M_1 M_2...M_n \) with their corresponding type such that \( n>2 \), and the parent node in the tree has the label \( \pi M_1 M_2...M_n \), annotate the parent node with the corresponding number of the derivation rule of the language and type generated. If no derivation rule is found or types are null, exit.

   - \( ND \) = nodes corresponding to one level higher of the tree.

Some examples of typed FOL \( \lambda \)-calculus formulas are the following:
\[
\lambda y. (\lambda x. (y \land (x@Joe))) : (t \rightarrow ((e \rightarrow t) \rightarrow t)) : r_4 \\
\lambda x. (y \land (x@Joe)) : ((e \rightarrow t) \rightarrow t) : r_4 \\
y \land (x@Joe) : t : r_5 \\
y : t \\
x@Joe : t : r_3 \\
x : (e \rightarrow t) \\
Joe : e
\]

Figure 1. Annotated derivation tree for the first typed FOL $\lambda$-calculus example formula.

**Example 14:**

- $\lambda y. \lambda x. (y \land (x@Joe))$ with type $(t \rightarrow ((e \rightarrow t) \rightarrow t))$ where $y$ has type $t$ and $x$ has type $e \rightarrow t$.

- $\lambda y. \lambda v. (\text{person}(y) \rightarrow (\text{man}(y) \lor v))$ with type $(e \rightarrow (t \rightarrow t))$ where $y$ has type $e$ and $v$ has type $t$.

- $\lambda v. (v \rightarrow \neg v)$ with type $(t \rightarrow t)$ where $v$ has type $t$.

- $\lambda w. (\lambda u. (w@\lambda z. (\text{loves}(z,u))))$ with type $(((e \rightarrow t) \rightarrow t) \rightarrow (e \rightarrow t))$ where $w$ has type $((e \rightarrow t) \rightarrow t)$ and $u$ has type $e$.

The derivation tree of the first example annotated with types and rules is shown in figure 1. One is given the variables and constants of the signature used to form the expression. In this case $Joe$ is a constant of type $e$, $y$ is a variable of type $t$ and $x$ is a variable of type $(e \rightarrow t)$.

The derivation tree of the fourth example annotated with types and rules is shown in figure 2. Again, one is given the variables and constants of the signature used to form the
expression. In this case variables $z$ and $u$ have type $e$ and variable $w$ has type $((e \rightarrow t) \rightarrow t)$.

The following are not typed FOL $\lambda$-calculus formulas:

**Example 15:**

- $\lambda y. (y \land \lambda x.(x@Joe))$
- $\lambda v. (v \rightarrow \neg v@Joe)$

In the first expression, suppose the term $Mary$ is applied to it. $(Mary \land \lambda x.(x@Joe))$ is obtained. If the term $\lambda v.person(v)$ is applied to this expression, one obtains: $(Mary \land \lambda x.(x@Joe))@\lambda v.person(v)$, where one can see that this is not a $\beta$-redex. A $\beta$-redex is a term of the form $(\lambda x.M)@N$, in this case the scope of $\lambda x$ is $(x@Joe)$ therefore a valid $\beta$-redex would be $(\lambda x.(x@Joe))@\lambda v.person(v)$ which is equivalent to $(\lambda x.M)@N$. But in this case $M = (Mary \land \lambda x.(x@Joe))$ has no $\lambda$-abstractor to form a $\beta$-redex. Thus, this expression cannot be reduced to a first-order logic formula. This expression is violating the fifth rule of FOL typed term construction where $\lambda x.(x@Joe)$ does not have type $t$. 

---

**Figure 2.** Annotated derivation tree for the fourth typed FOL $\lambda$-calculus example formula.
\[
\lambda y.(y \land \lambda x.(x@Joe))
\]
\[
(\: y \land \lambda x.(x@Joe)\:)
\]
\[
y : t \quad \lambda x.(x@Joe) : ((e \rightarrow t) \rightarrow t) : r_4
\]
\[
x@Joe : t : r_3
\]
\[
x : (e \rightarrow t) \quad Joe : e
\]

Figure 3. Annotated derivation tree for the first typed FOL \(\lambda\)-calculus example that is not a formula.

The derivation tree for this example is shown on figure 3.

The second expression has occurrences of two \(v\) variables which have different type. The type of the \(v\) before the right arrow is \((t \rightarrow t)\) and the type of the second \(v\) after the right arrow is \((e \rightarrow t)\), since \(Joe\) has type \(e\). The abstractor \(\lambda v\) binds to the variable that \(v\) is representing in the abstractor. This variable can only have one type, but not both. The two \(v\) variables of the expression are different symbols of the signature. Therefore, one of the variables is not bound and the expression does not follow the definition of typed FOL \(\lambda\)-calculus formulas.

The sets \(D_a\) and \(\Delta_a\) presented above, may look similar but they should not be confused. \(\Delta_a\) is the set of FOL typed terms of type \(a\) that can be constructed using symbols of the signature. For example, for type \(t\), the typed terms \(person(John) \land man(John)\) or \(woman(Mia)\) belong to the set \(\Delta_t\); while the elements of the set \(D_t\) are truth values, \{true, false\}, which describe the possible meanings of the typed terms of \(\Delta_t\). \(woman(Mia)\) as a first-order formula, can have the meaning of true or false. Every FOL typed term is a well-formed expression of the language, of type \(a\), whose semantics
comply with the set $D_a$ of possible denotations of expressions of type $a$.

Consider the typed FOL formula $\lambda x.\text{person}(x)$, which has type $e \rightarrow t$. In this case, $x$ is a typed variable ranging over the entities of the domain and, therefore, it has type $e$. When an individual from the domain, like the typed constant of type $e$: “John”, is applied to the formula, a first-order formula $\text{person}(\text{John})$ is obtained, which is of type $t$. The formula is no more than a function from individuals to truth values.

Now, with the signature for Typed FOL $\lambda$-calculus, one can specify a model with a domain of entities and a function to assign semantics to elements of the signature. With this model, one can choose an interpretation for the formula $\lambda x.\text{person}(x)$ where only those entities of the domain that belong to the set of the predicate $\text{person}$ would return the value $true$ as output of the function. Therefore, the interpretation of $\lambda x.\text{person}(x)@\text{John}$ would be the same as the one for $\text{person}(\text{John})$ in the model. This is assured by the well-typed application that is taking place between the formula $e \rightarrow t$ and the argument $e$.

In a well-typed application, the type of the argument is the same as the “input” type of the function. Types are in charge of regulating which applications are possible and when both argument and function have the correct types, one has a well-typed expression.

This section concludes with two more definitions. Since the signature and term definition in typed FOL lambda calculus is different from the one presented for classic lambda calculus, the concept of occurrence needs to be re-defined.

**Definition 27** [occurrence] The relation $P$ occurs in $Q$ is defined by induction on $Q$ as follows:
• a FOL typed term P occurs in P.

• if P occurs in M or in N, then P occurs in M@N.

• if P occurs in M, then P occurs in \( \lambda x.M \).

• if P occurs in \( \phi \) or P occurs in \( \psi \), then P occurs in \( \phi \lor \psi \), \( \phi \land \psi \), \( \phi \rightarrow \psi \).

• if P occurs in \( \phi \), then P occurs in \( \neg \phi \).

• if P occurs in \( \phi \), then P occurs in \( \forall u \phi \) and \( \exists u \phi \).

• if P occurs in \( \phi \) or P occurs in \( \psi \), then P occurs in \( \phi = \psi \).

• if P occurs in any typed term \( t \), then P occurs in \( f(t_1, \ldots, t_n) \). Where \( f \) is a predicate symbol or a function symbol.

**Example 16:** Consider the typed FOL \( \lambda \)-calculus formula \( J = \lambda w.\lambda u.(woman(Mary) \land w@\lambda z.(loves(z, u))) \). The typed terms \( loves(z, u) \), \( woman(Mary) \), \( \lambda z.(loves(z, u)) \) and \( w \) occur in \( J \).

**Definition 28 [FOL \( \lambda \)-component]** Constants, quantifier variables, connectives \( \neg \), \( \land \), \( \lor \) and \( \rightarrow \), quantifiers \( \exists \) and \( \forall \), predicates and function symbols, and the equality symbol \( = \) are denoted as FOL \( \lambda \)-components. The set of FOL \( \lambda \)-components of a formula \( J \) is identified as \( LC(J) \).

Basically, all FOL \( \lambda \)-elements except for the lambda application symbol, lambda abstractors and their corresponding bound variables, are considered FOL \( \lambda \)-components.

**Example 17:** Consider the typed FOL \( \lambda \)-calculus formula \( J = \lambda w.\lambda u.(woman(Mary) \land w@\lambda z.(loves(z, u))) \). \( LC(J) = \{\text{woman, Mary, } \land, \text{loves}\} \).
2.2. Typed Answer Set Programming Lambda Calculus

Answer Set Programming is a declarative language, belonging to the class of logic programming languages which use the answer set semantics. As stated in [15], this language is one of the most suitable declarative language for knowledge representation and reasoning. It has more support from the AI research community than any other Knowledge Representation language, including efficient and live implementations and theoretical building block studies. This language also allows the representation, in an intuitive way, of various kinds of knowledge that cannot be adequately expressed in first-order logic. These include, for instance, default statements (most birds fly) and normative statements (normally birds fly).

In the real world, knowledge of nature and the surroundings is normally expressed in terms of defaults. Everything behaves as one expects unless some exception or special condition changes the environment. Therefore, many sciences express their knowledge assuming that objects and situations function in a “normal” or default way and thus some properties hold. But, in the presence of a different setting, the same properties may no longer be true. Since representation of knowledge in many domains requires the ability to express defaults and normative statements, this work contributes towards automatically translating natural language text to Answer Set Programming theories.

The same process as in the previous section will be followed, by presenting the signature for the language Typed Answer Set Programming Lambda Calculus (Typed ASP Lambda Calculus). Then, in order to obtain formal results, one needs to present the same definitions introduced in the previous section for the case of Typed ASP Lambda Calculus.
The signature of the language is defined as follows:

- The lambda operator $\lambda$
- The lambda application $\odot$
- The parenthesis $\left(, \right), [,$
- For every type $a$, an infinite set of variables $v_{n,a}$ for each natural number $n$
- For every type $a$, a (possibly empty) set of constants $c_a$ of type $a$
- The connectives $\text{or}$, $\leftarrow$, $\neg$, $\text{not}$ and "",
- Predicate symbols with a given arity $n$

**Definition 29** [ASP $\lambda$-element] Symbols of the typed ASP logic lambda calculus signature will be denoted as ASP $\lambda$-elements.

Next, definitions related to Answer Set Programming syntax and semantics are presented. The theory from [15] is followed.

**Definition 30** An atomic term is either a variable or a constant.

**Definition 31** An atomic term is ground, if it is a constant.

**Definition 32** The Herbrand Universe of a language is the set of all ground terms which can be formed with the constants of the language.

**Definition 33** An atom is of the form $f(t_1, \ldots, t_n)$ where $f$ is a predicate symbol and each $t_i$ is an atomic term. If each $t_i$ is ground, then the atom is ground.

**Definition 34** The Herbrand Base of a language is the set of all ground atoms.
Definition 35 A literal is an atom or an atom preceded by the unary connective $\neg$. They are called positive literal and negative literal respectively. A literal is ground if the atom is ground.

Definition 36 A gen-literal is a literal or a literal preceded by the unary connective not.

Definition 37 An ASP rule is a combination of literals and connectives like the following:

$$L_0 \lor \ldots \lor L_k \leftarrow L_{k+1}, \ldots, L_m, \lnot L_{m+1}, \ldots, \lnot L_n.$$  

where $L_i$ are literals and $k \geq 0$, $m \geq k$ and $n \geq m$.

An intuitive meaning of the rules can be expressed as follows: If $L_{k+1}, \ldots, L_m$ are true and for $L_{m+1}, \ldots, L_n$ one has no reason to believe that they are true, then at least one of $L_0$ or $\ldots$ or $L_k$ has to be true. The literals to the left of the “←” belong to the Head of the rule, and the literals to the right of the “←” belong to the Body of the rule.

Rules with an empty Head are called Constraints. They are written as:

$$\leftarrow L_{k+1}, \ldots, L_m, \lnot L_{m+1}, \ldots, \lnot L_n.$$  

Rules with an empty body and one literal in the head are called Facts. They are written without the arrow as:

$$L_0.$$  

An ASP program is a set of ASP rules.

Example 18: Consider the set of literals $\{h(a), h(b), h(c)\}$. An example of an ASP rule would be:

$$h(a) \rightarrow h(b), \lnot h(c).$$
An example of a constraint would be:

\[ \rightarrow h(b), h(c). \]

And an example of a fact would be:

\[ h(b). \]

**Definition 38** A *Herbrand Interpretation* of an ASP program \( \Pi \) is any subset \( I \) of the Herbrand Base of the program \( HB_\Pi \).

**Definition 39** A *Partial Herbrand Interpretation* of an ASP program \( \Pi \) is any subset \( I \) of the set of all literals of the program \( \Pi \).

**Definition 40** [Satisfiability] An *ASP rule* of the form:

\[ L_0 \text{ or } ... \text{ or } L_k \leftarrow L_{k+1}, ..., L_m, \text{ not } L_{m+1}, ..., \text{ not } L_n. \]

of an ASP program \( \Pi \) is said to be *satisfied* by a Partial Herbrand Interpretation \( I \) of \( \Pi \) if:

1. \( k = 0 \) and \( L_0 \) is empty:
   
   \( \{L_{k+1}, ..., L_m\} \not\subseteq I \text{ or } \{L_{m+1}, ..., L_n\} \cap I \neq \emptyset. \)

2. Otherwise:

   \( \{L_{k+1}, ..., L_m\} \subseteq I \text{ and } \{L_{m+1}, ..., L_n\} \cap I = \emptyset \) implies that \( \{L_0, ..., L_k\} \cap I \neq \emptyset. \)
**Example 19:** Consider the following:

- The rule \( h(a) \rightarrow h(b), \text{not } h(c) \).
- A Partial Herbrand Interpretation \( I = \{ h(a), h(b) \} \).

The rule is satisfied since:

- \( h(b) \subseteq I \).
- \( h(c) \cap I = \emptyset \).
- \( h(a) \cap I \neq \emptyset \).

**Example 20:** Consider the following:

- The rule \( h(a) \rightarrow h(b), \text{not } \neg h(c) \).
- A Partial Herbrand Interpretation \( I = \{ \neg h(c), h(b) \} \).

In this case, the rule is trivially satisfied since:

- \( \neg h(c) \cap I \neq \emptyset \).

**Definition 41** A Partial Herbrand Model of an ASP Program \( \Pi \) is a Partial Herbrand Interpretation \( I \) of \( \Pi \) that satisfies all rules in \( \Pi \).

**Definition 42** An Answer Set of an ASP Program \( \Pi \) without the “not” operator, is a partial Herbrand Model of \( \Pi \) that is minimal among all the possible partial Herbrand Models of \( \Pi \).

**Definition 43** [reduct of an ASP program] Let \( \Pi \) be an ASP program without variables. For any set \( S \) of literals, let \( \Pi^S \) be the ASP program without the “not” operator, obtained from \( \Pi \) by deleting
1. each rule that has a gen-literal not L in its body with L ∈ S

2. all gen-literals of the form not L in the bodies of the remaining rules

**Example 21:** Consider the ASP program Π:

\[ h(a) \rightarrow h(b), \text{not } h(c). \]
\[ h(b) \rightarrow \text{not } h(a). \]
\[ h(a) \rightarrow \text{not } h(b). \]

Consider the set \( S = \{h(a)\} \), the reduct \( Π^S \) is:

\[ h(a) \rightarrow h(b), h(a). \]

\( Π^S \) is an ASP Program without the “not” operator and its answer sets were previously defined. The definition of answer set for any ASP program is as follows.

**Definition 44** A set \( S \) of literals is an Answer Set of an ASP Program \( Π \) if \( S \) belongs to the set of answer sets of \( Π^S \).

**Example 22:** Consider the ASP program \( Π \) presented in the last example. \( Π \) has only one answer set which is \( S = \{h(a)\} \). This is because \( S \) belongs to the answer sets of the reduct of \( Π \) shown in the previous example.

Next, the set of types that will be used with Typed ASP Lambda Calculus, in conjunction with the definition of the semantics of types assigned to the different expressions of the language is introduced. The principles presented in [12] will again be followed, where \( D_a \) represents the set of possible objects (denotations) that describe the meanings of expressions of type \( a \).
**Definition 45** The set of types $\Theta$ is defined recursively as follows:

1. $e$ is a type
2. $l$ is a type
3. $h$ is a type
4. $d$ is a type
5. $t$ is a type
6. If $a$ and $b$ are types, then $(a \rightarrow b)$ is a type.

**Definition 46** The semantics are defined as:

- $D_e$ = the set of atomic terms.
- $D_l$ = the set of gen-literals.
- $D_h$ = the set of “or”-connected literals that belong to heads of ASP rules.
- $D_d$ = the set of “,”-connected gen-literals that belong to bodies of ASP rules.
- $D_t$ = $\{0, 1\}$, the set of satisfiability values for an ASP rule.
- $D_{a \rightarrow b}$ = the set of functions from $D_a$ to $D_b$.

Expressions of type $t$ denote satisfiability values of rules of an ASP program. An ASP rule can be true under certain Herbrand interpretations, and false under others. $(a \rightarrow b)$

---

2 A set of “or”-connected literals is defined as the set of literals separated by a connective $\oplus$ in all possible combinations. For example, for the set $\{x, y\}$, the “or”-connected set is $\{x \oplus y, y \oplus x\}$. 
denotes functions which input is in $D_a$ and output values are in $D_b$. For example, the type $e \to s$ corresponds to functions from atomic terms to satisfiability values.

This section continues by introducing the definition for ASP typed term, followed by the definition of ASP $\lambda$-calculus formula:

**Definition 47** [ASP typed term] The elements which belong to the set $\Theta_\alpha$ of ASP typed terms of type $\alpha$ are inductively defined as follows:

1. For each type $a$, every atomic term of type $a$ belongs to $\Delta_a$. Atoms belong to $\Delta_l$.

2. For any types $a$ and $b$, if $\alpha \in \Delta_{a \to b}$ and $\beta \in \Delta_a$, then $\alpha @ \beta \in \Delta_b$.

3. For any types $a$ and $b$, if $u$ is a variable of type $a$ and $\alpha \in \Delta_b$ and has free occurrences of the variable $u$, then $\lambda u. \alpha \in \Delta_{a \to b}$ and the free occurrences of $u$ are now bound to the abstractor $\lambda u$.\(^3\)

4. If $\alpha \in \Delta_l$ or $\in \Delta_h$ and $\beta \in \Delta_d$ or $\in \Delta_l$, then $(\alpha \leftarrow \beta) \in \Delta_t$.

5. If $\alpha \in \Delta_d$ or $\in \Delta_l$, then $(\leftarrow \alpha) \in \Delta_l$.

6. If $\alpha \in \Delta_l$ and not $\beta$ does not occur in $\alpha$, then $(\text{not } \alpha) \in \Delta_l$.

7. If $\alpha \in \Delta_l$ and $\neg \beta$ or not $\beta$ does not occur in $\alpha$, then $(\neg \alpha) \in \Delta_l$.

8. If $\alpha \in \Delta_h$ or $\in \Delta_l$ and $\beta \in \Delta_l$ or $\in \Delta_h$ and not $\gamma$ does not occur in $\alpha$ and $\beta$, then $(\alpha \text{ or } \beta) \in \Delta_h$.

9. If $\alpha \in \Delta_d$ or $\in \Delta_l$ and $\beta \in \Delta_l$ or $\in \Delta_d$, then $(\alpha, \beta) \in \Delta_d$.

\(^3\)Refer to the definition of occurrence presented at the end of this section.
**Definition 48** [ASP λ-calculus formula] A typed ASP λ-calculus formula is an ASP typed term of type $a$ where every variable is bound to an abstractor and every abstractor binds to a variable.

These conditions ensure that one obtains Answer Set Programming programs when the typed ASP λ-calculus formulas are in β-normal form. These two conditions correspond to closed and λI-terms defined in the previous chapter of lambda calculus. In order to use the derivation tree algorithm presented in the previous section, some definitions for typed ASP lambda calculus need to be specified as follows:

**Definition 49** [unary ASP λ-elements] The unary ASP λ-elements, identified as $v$, correspond to the lambda operator $λ$, the connectives $\neg$, $not$.

**Definition 50** [binary ASP λ-elements] The binary ASP λ-elements, identified as $ω$, correspond to the lambda application $@$, the connectives $or$, $←$ and $\cdot$.

**Definition 51** [n-ary ASP λ-elements] The n-ary ASP λ-elements, identified as $ϖ$, correspond to predicate symbols of the signature.

Some examples of typed ASP λ-calculus formulas are the following:

**Example 23:**

- $λw.λv.(w ← v@X.)$ with type $(h → ((e → d) → t))$ where $w$ has type $h$ and $v$ has type $e → d$.

- $λx.λy.(← h(x), not − y)$ with type $(e → (l → t))$ where $x$ has type $e$ and $y$ has type $l$.

- $λv.(v or ¬v.)$ with type $(l → t)$ where $v$ has type $l$. 

34
\( \lambda w. (\lambda u. (w@\lambda v. (\text{position}(v, u)))) : (((e \to l) \to t) \to (e \to t)) : r_4 \)

\[ \lambda u. (w@\lambda v. (\text{position}(v, u))) : (e \to t) : r_4 \]

\[ w@\lambda v. (\text{position}(v, u)) : t : r_5 \]

\[ w : ((e \to l) \to t) \quad \lambda v. (\text{position}(v, u)) : (e \to l) : r_3 \]

\[ \text{position}(v, u) : l : r_1 \]

\[ v : e \\
\]

\[ u : e \]

Figure 4. Annotated derivation tree for the fourth typed ASP \( \lambda \)-calculus example.

- \( \lambda w. (\lambda u. (w@\lambda v. (\text{position}(v, u)))) \) with type \(((e \to l) \to t) \to (e \to t))\) where \( w \) has type \(((e \to l) \to t)\) and \( u \) has type \( e \).

The derivation tree for the fourth example formula is shown in figure 4. Once again, one is given the variables and constants of the signature used to form the expression. In this case variables \( v \) and \( u \) have type \( e \) and variable \( w \) has type \( ((e \to l) \to t) \).

Let the fourth formula of the example be \( J \). In \( J \), \( w \) has type \( ((e \to l) \to t) \) because when an ASP typed formula is applied \( J \), it will be placed in the variable \( w \) and will receive as argument the expression \( \lambda v. (\text{position}(v, u)) \). This expression has type \( e \to l \) and therefore the input of the formula applied to \( J \) needs to have \( e \to l \) as input and \( t \) as output to lead to an ASP formula. Thus, \( w \) has type \( ((e \to l) \to t) \). \( u \) has type \( e \) meaning that one expects an atomic term to be placed inside the literal \( \text{position} \).

The following are not typed ASP \( \lambda \)-calculus formulas:

1. \( \lambda y. \lambda x. (y \text{ or } not \ x@X) \)

2. \( \lambda v. \lambda w. (\neg w \iff \neg not \ v@X) \)
The first expression has an occurrence of an “or” connective which can only appear in the head of formulas. On the other hand, there is an occurrence of the connective not which can only appear in the body of formulas. Therefore, the expression does not follow the rules of typed ASP $\lambda$-calculus formulas.

The second expression violates the seventh rule of typed ASP $\lambda$-calculus formulas since the connective $\neg$ can only be applied to typed terms of type $l$ with no occurrences of $\text{not}\beta$. In this case, $\text{not}\beta$ occurs as “$\neg v@X$”. The derivation tree for this example can be found on figure 5.

As discussed in the previous section, the sets $D_a$ and $\Delta_a$ should be regarded as different. In this case, $\Delta_a$ is the set of ASP typed terms of type $a$ that can be constructed using symbols of the signature. For example, for type $d$, the typed term $\text{bird}(X)$, $\neg \text{fly}(X)$ formed by symbols of the signature, belongs to the set $\Delta_d$; while the elements of the set $D_d$ represent the possible bodies of ASP rules formed by sets of literals connected by the “,” connective of the language. Therefore, the previous expression built with symbols of
the signature describes a possible body of an ASP rule formed by gen-literals “bird(X)” and “not¬fly(X)”.

To obtain answer sets for ASP programs, one sets up a model for the language that consists of the signature of typed ASP lambda calculus and a Partial Herbrand Interpretation. The interpretation indicates the subset of literals from the signature of the program that are considered to calculate the satisfiability value of the rules of a program. Consider the formula:

\[ \lambda x.\lambda y.(h(a) \leftarrow x, \neg h(y)). \]

which has type \((d \rightarrow (e \rightarrow t))\). In this case, a typed term of type \(d\) can, for example, be “\(h(b), h(c)\)”. When applied to the formula, one obtains an expression with type \(e \rightarrow t\). This formula will accept as input an atomic term of the language, for example “\(a\)”, that will be placed in the atom \(h()\). After the application of the atomic term to the example formula, one obtains the following ASP rule of type \(s\):

\[ h(a) \leftarrow h(b), h(c), \neg h(a)). \]

At this point, one can take the Partial Herbrand Interpretation from the model and calculate the satisfiability value of the rule.

This section concludes with two more definitions. As in the previous section, the definition for occurrence needs to be redefined for this language:

**Definition 52** [occurrence] The relation \(P\) occurs in \(Q\) is defined by induction on \(Q\) as follows:

- an ASP typed term \(P\) occurs in \(P\).
• if P occurs in M or in N, then P occurs in M@N.

• if P occurs in M, then P occurs in \( \lambda x. M \).

• if P occurs in \( \phi \) or P occurs in \( \psi \), then P occurs in \( \phi \text{ or } \psi \), \( \phi \leftarrow \psi \) and \( \phi \), \( \psi \).

• if P occurs in \( \phi \), then P occurs in \( \neg \phi \) and not \( \phi \).

• if P occurs in any atomic term \( t_i \), then P occurs in \( P(t_1, ..., t_n) \). Where \( P \) is a predicate symbol of an atom.

**Example 24:** Consider the typed ASP \( \lambda \)-calculus formula \( J = \lambda x. \lambda y. (h(a) \text{ or } h(y) \leftarrow h(x), \neg y) \). The typed terms \( h(a) \text{ or } h(y) \), \( \neg y \), \( \lambda x. \lambda y. (h(a) \text{ or } h(y) \leftarrow h(x), \neg y) \) occur in \( J \).

**Definition 53 [ASP \( \lambda \)-component]** Constants, predicate symbols, connectives or, \( \leftarrow \), \( \neg \), not and “,” are denoted as ASP \( \lambda \)-components. The set of ASP \( \lambda \)-components of a formula \( J \) is identified as LC(J).

Basically, all ASP \( \lambda \)-elements except for the lambda application symbol, lambda abstractors and their corresponding bound variables, are considered ASP \( \lambda \)-components.

**Example 25:** Consider the typed ASP \( \lambda \)-calculus formula \( J = \lambda x. \lambda y. (h(a) \text{ or } h(y) \leftarrow h(x), \neg y) \). LC(J) = \{h, a, or, \leftarrow, not, \neg\}.

**2.3. Type Order**

So far, typed \( \lambda \)-calculus has been introduced and, for the two languages, the different types that will be assigned to different typed terms were defined. Now the notion of order is defined which is associated with these types and that separates typed \( \lambda \)-calculus formulas.
to several classes. The notion of orders is used to show that the Inverse $\lambda$-Algorithms are complete algorithms for typed $\lambda$-calculus formulas up to order two.

Each typed term has a type, and each type will be assigned an order. Next, the definition of type order is presented and some intuition behind it:

**Definition 54** [Order of a type] The order of a type is defined as:

1. Base types have order 0.

2. For function types, $\text{order}(a \rightarrow b) = \max(\text{order}(a) + 1, \text{order}(b))$.

**Example 26:** In the case of typed FOL lambda calculus, base types are $e$ and $t$. Function types of first order are $(e \rightarrow t)$ or $(e \rightarrow (e \rightarrow t))$. A function type of second order is $(((e \rightarrow t) \rightarrow t) \rightarrow t)$ and one of third order is $((((e \rightarrow t) \rightarrow t) \rightarrow t) \rightarrow t)$.

To provide an intuition of the definition, consider the binary tree of a type built from the “$\rightarrow$”. The order is the depth of the most left-nested arrow, starting from one. Meaning, how long can one maximally go left in the tree (not necessarily in consecutive steps) before one reaches a leaf.  

Some examples of typed FOL lambda calculus formulas of different orders will be presented so that the difference between the different orders is more easily understandable:

**Example 27:**

- Typed FOL lambda calculus formulas of order zero. Formulas which type is a base type.

  - *Mia* - type $e$.

---

4Personal Communication with Andreas Rossberg - Max Planck Institute for Software Systems (Type Systems and Functional Programming Group) and Thorsten Altenkirch - University of Nottingham (Functional Programming Laboratory)
- $Vincent$ - type $e$.
- $woman(Mia)$ - type $t$.
- $\exists x(loves(Vincent, x))$ - type $t$.
- $John$ - type $e$.
- $\exists x(plane(x) \land takes(John, x))$ - type $t$.
- $man(Vincent) \land person(Vincent)$ - type $t$.
- $loves(Mia, Vincent)$ - type $t$.

- Typed FOL lambda calculus formulas of order one.

- $\lambda v. v$ - type $(t \to t)$. (One of the possible types)
- $\lambda v. (loves(Mia, v))$ - type $(e \to t)$.
- $\lambda v. \lambda u. (\exists x(plane(u) \land takes(v, u)))$ - type $(e \to (e \to t))$.
- $\lambda v. \lambda w. \lambda u. (v \lor (w \to u))$ - type $(t \to (t \to (t \to t)))$.
- $\lambda v. \lambda u. (\forall x(man(v) \to u))$ - type $(e \to (t \to t))$.
- $\lambda v. (alive(Vincent) \to v)$ - type $(t \to t)$.
- $\lambda v. (\exists v(plane(v) \land takes(John, v)))$ - type $(e \to t)$.

- Typed FOL lambda calculus formulas of order two.

- $\lambda v. v$ - type $((e \to t) \to (e \to t))$. (One of the possible types)
- $\lambda v. (v @ loves(Mia, Vincent))$ - type $((t \to t) \to t)$.
- $\lambda v. \lambda u. (man(v) \to (u @ Mia))$ - type $(e \to ((e \to t) \to t))$.
- $\lambda v. (v @ Vincent)$ - type $((e \to t) \to t)$.
- $\lambda v.\lambda u.(\exists x(v \land u@woman(Mia)))$ - type $(t \to ((t \to t) \to t))$.

- $\lambda v.\lambda u.(v@Vincent \land u@Mia)$ - type $((e \to t) \to ((e \to t) \to t))$.

- $\lambda u.(person(Mia) \lor u@Mia)$ - type $((e \to t) \to t)$.

- $\lambda u.\lambda v.(\exists x(u@x \land v@x))$ - type $((e \to t) \to ((e \to t) \to t))$.

- Typed FOL lambda calculus formulas of order three.

- $\lambda v.v$ - type $(((e \to t) \to t) \to ((e \to t) \to t))$. (One of the possible types)

- $\lambda v.(v@\lambda u.(loves(u, Vincent)))$ - type $(((e \to t) \to t) \to t)$.

- $\lambda v.\lambda u.(\exists x(v@\lambda w.(w \land man(Vincent)) \land u@\lambda y.(woman(y))))$ - type $(((t \to t) \to t) \to ((e \to t) \to t))$.

- $\lambda w.\lambda v.(man(w) \land v@\lambda u.(loves(u, v)))$ - type $e \to (((e \to t) \to t) \to t))$.

- $\lambda w.\lambda v.\lambda t.(w@Mia) \land v@\lambda u.(loves(u, t))$ - type $((e \to t) \to (((e \to t) \to t) \to t))$.

- $\lambda v.(v@\lambda u.\lambda w.(loves(u, w)))$ - type $(((e \to (e \to t)) \to t) \to t)$.

- $\lambda w.\lambda v.(v@Vincent) \land w@\lambda u.(loves(u, Mia))$ - type $((e \to t) \to t) \to ((e \to t) \to t))$.

With these examples, one can see the intuition behind the order of typed FOL lambda calculus formulas. Formulas of order zero correspond to expressions with base types. Formulas of type $e$ represent individuals of the domain and formulas of type $t$ represent first-order formulas. Formulas of order one correspond to expressions which start with a series of lambda abstractors followed by first-order theory expressions with variables bound to the initial lambda abstractors.
Formulas of order two extend the expressions allowed in order one by including applications. Formulas of order zero with type \( e \) or \( t \) can be applied to variables inside the formula. Formulas of order three extend those present in order two by allowing lambda abstractors inside the expression after the initial lambda abstractors. In this case, formulas of order one can be applied to variables, this is why now, we can find lambda abstractors at the beginning and in the middle of the formulas.

In these examples, the term \( \lambda x. x \) was included and one can see that it can have several orders. This term acts like a neutral operator of lambda calculus. When a term \( M \) is applied to \( \lambda x. x \), the result is the same term \( M \). Therefore, depending on the order of \( M \), \( \lambda x. x \) will have different orders. Note that, the input type and output type of \( \lambda x. x \) is always the same to allow the described behavior.

Next, an example of applying the definition of order of a type is introduced. Consider the third order type \(((e \rightarrow t) \rightarrow t) \rightarrow t \). One can annotate the order of the different types following the definition presented earlier. One can do this by underscoring each of the types and function types with its order adding them as one moves inside the function type following the definition.

**Example 28:**

\[
((e_{1+1+1} \rightarrow t_{0+1+1})_{1+1} \rightarrow t_{0+1})_1 \rightarrow t_0
\]

The order of the function type of the example is the maximum value of its orders, order three, the value of the most inner type. If one builds the tree, one can easily see that the type with the number three would be the leaf to which one can go down the tree, three times to the left. Figure 6 shows the tree representation for the above example.
Figure 6. Binary tree representation for a third order type where the depth of the most left-nested arrow is three.

Figure 7. Binary tree representation for a second order type where the depth of the most left-nested arrow is two.

Consider the following second order type:

**Example 29:**

\[((e \rightarrow t) \rightarrow (e \rightarrow (e \rightarrow t)))\]

If one builds the binary tree representation of this example starting at the root of the tree and going down the left side of the tree, a leaf is reached after two nodes. Therefore, the arrow in the left subtree, is the most left-nested arrow at level two and sets order two for the type. Figure 7 shows the tree for this example.

With this definition for order and the types from typed FOL lambda calculus, the following can be stated:

- First order types receive a base type as input and return:
– a base type, for instance: \( e \rightarrow t \).

– a first order type, for instance: \( e \rightarrow (e \rightarrow t) \).

• Second order types receive a base type or a function as input and return:

– a base type, for instance: \( (e \rightarrow t) \rightarrow t \).

– a first order type, for instance: \( (e \rightarrow t) \rightarrow (e \rightarrow t) \).

– a second order type, for instance: \( e \rightarrow ((e \rightarrow t) \rightarrow t) \).

The Inverse \( \lambda \)-Algorithms manipulate three different typed \( \lambda \)-calculus formulas \( F, G \) and \( H \). To develop the completeness proof, one needs to look at the possible orders that these three formulas may have. This is how, when developing the proofs, it will be known how many cases will need to be handled.

Thus, there are initially 27 possible combinations of orders for these three formulas from order zero to two. If one considers \( H = F \circledcirc G \) or \( H = G \circledcirc F \), however, most of these possible combinations are not valid anymore for the following reasons (consider the first case):

**Lemma 1.** If \( H = F \circledcirc G \), then the formula \( F \), which is receiving \( G \) as input, needs to have at least one order more than \( G \).

*Proof.* Input type is located to the left of the type constructor \( \rightarrow \) of \( F \) and by the definition of order for function types, \( F \) will have at least one more order than its input type. \( \square \)

**Lemma 2.** If \( H = F \circledcirc G \), then the output type of the formula \( F \) cannot have a higher order than \( H \). Thus, \( H \) will always have same or less order than \( F \).
Proof. H is the output type of F therefore, since the type of H is part of the type of F, H cannot be higher than F because that would change the type of F.

Lemma 3. If $H = F \circ G$, then if F is order two and H is order zero, G needs to be order one in order to obtain a formula of order zero as output.

Proof. H is the output type of F. H is order zero, therefore, in order for F to be order two, the input type needs to be at least order one. In this way, one obtains an order two type for F following the definition of order.

Lemma 4. If $H = F \circ G$, then if F is order two and H is order one, then G needs to be order one.

Proof. If G is order zero, F cannot be order two since input is G, order zero, and output is H, order one, which generates an order one type. Consider G, an input type of order zero, for example, e, and H, an output type of order one, for example, $(e \rightarrow t)$, then F is $(e \rightarrow (e \rightarrow t))$ which is order one by definition. If G is order two, then F cannot be order two. Since G is the input type of F, it would transform F into a third order type.

Considering these four lemmas, F cannot be order zero, since it needs to receive G as input. If F is order one, G can only be order zero and H can only be order zero or one. If F is order two, G can be order zero or one, and H can be any order; as long as condition 1 and 3 above are satisfied. In total, there are 6 possible order combinations for F, G and H. These combinations are shown in table 2 with examples from the types of typed FOL lambda calculus. One can observe that the input type of F corresponds to the type of G and the output type of F corresponds to the type of H.
### Table 2

Possible order combinations for F, G and H formulas.

<table>
<thead>
<tr>
<th>H</th>
<th>F</th>
<th>G</th>
<th>FOL examples for formula F</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>$e \rightarrow t$</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>$e \rightarrow (e \rightarrow t)$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>0</td>
<td>$e \rightarrow ((e \rightarrow t) \rightarrow t)$</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>1</td>
<td>$(e \rightarrow t) \rightarrow t$</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>$(e \rightarrow t) \rightarrow (e \rightarrow t)$</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>$(e \rightarrow t) \rightarrow ((e \rightarrow t) \rightarrow t)$</td>
</tr>
</tbody>
</table>

2.4. The Inverse Lambda Algorithms

This section commences by discussing some lemmas and properties that are important for the Inverse $\lambda$-Algorithms and that are important to understand their completeness. The lemmas show characteristics of classic $\lambda$-calculus in relation with the typed $\lambda$-calculus that was defined in previous sections. After this introduction, two Inverse $\lambda$-Algorithms will be defined along with the explanation of their different parts. First, a lemma based on lemma 1B1.1 from [11] is presented.

**Lemma 5.** Given typed $\lambda$-calculus formulas $H$, $G$ and $F$, if $G@F \ \beta$-contracts to $H$, then $\text{LC}(G@F) = \text{LC}(H)$\(^5\).

**Proof.** The definition of typed $\lambda$-calculus formulas presented in previous sections states that all variables that appear in a formula are bound. In the case of typed FOL lambda calculus, if there are variables associated to quantifiers that are not bound in $G$, one will apply the necessary $\alpha$-conversions in case that variable clashes may occur. Therefore, when one formula is applied to another, none of the lambda components of those formulas

\(^5\text{LC was defined in the previous chapter as the set of lambda components of a formula}\)
will be modified by the application due to the definition of application in lambda calculus.

\[ \square \]

**Example 30:** Consider the formulas

\[ G = \lambda u.\lambda v. (person(v) \land loves(v, u)) \]

\[ F = \lambda w. ((w @ Mia @ Vincent) \lor (w @ Vincent @ Mary)) \]

and the formula

\[ H = (person(Vincent) \land loves(Vincent, Mia)) \lor (person(Mary) \land loves(Mary, Vincent)) \]

where \( H = F \@ G \):

- \( \text{LC}(G) = \{ \text{person}, \land, \text{loves} \} \)
- \( \text{LC}(F) = \{ \text{Mia}, \text{Vincent}, \lor, \text{Mary} \} \)
- \( \text{LC}(F@G) = \{ \text{person}, \land, \text{loves}, \text{Mia}, \text{Vincent}, \lor, \text{Mary} \} \)
- \( \text{LC}(H) = \{ \text{person}, \text{Vincent}, \land, \text{loves}, \text{Mia}, \lor, \text{Mary} \} \)

It is easy to see that \( F@G \beta\)-contracts to \( H \) and every \( \lambda \)-component of the set \( \text{LC}(F@G) \) is present in \( \text{LC}(H) \).

This lemma intuitively tries to show that: given typed \( \lambda \)-calculus formulas \( H, G \) and \( F \) such that \( H = G@F \) or \( H = F@G \), any \( \lambda \)-component of \( H \) must be contained in either \( F \) or \( G \) (since one obtains \( H \) from the application of these two formulas), and it cannot be the case that a \( \lambda \)-component in \( F \) or \( G \) will not appear in \( H \) (since there would be no proper second formula in the application that would give \( H \)). This characteristic of the structure of the application is an essential part of the Inverse \( \lambda \)-Algorithms and the way in which they construct the missing formula from the other two.
Since typed $\lambda$-calculus formulas are $\lambda I$ terms, the case where one sets $F$ to be $\lambda v.H$ is eliminated. This formula does not provide any semantic meaning to the expression that $F$ represents; but, it would be a valid classic lambda calculus formula for the application. The objective is to keep the semantic information between $F$ and $G$, which combined, give the semantics of $H$. By assuring that the information in $G$ will lead to obtain the semantics of $H$, $F$ is provided with a semantic representation that contains the desired value.

**Lemma 6.** Given typed $\lambda$-calculus formulas $H$ and $G$, if $G$ is a sub-term of $H$, then there is always an $F$ such that $H=F@G$.

*Proof.* Let $G = g_1, g_2, ..., g_n$. (where $g_1, ..., g_n$ are $\lambda$-elements)

We know that $G$ occurs in $H$. Thus,

Let $H = h_1, h_2, ..., h_i, g_1, ..., g_n, h_{i+1}, ..., h_n$.

Let $F$ be $\lambda v.h_1, h_2, ..., h_i, v, h_{i+1}, ..., h_n$.

Then, $H = h_1, h_2, ..., h_i, g_1, g_2, ..., g_n, h_{i+1}, ..., h_n = \lambda v.h_1, h_2, ..., h_i, v, h_{i+1}, ..., h_n @ g_1, g_2, ..., g_n = F@G$. □
3. INVERSE LAMBDA CALCULUS ALGORITHMS

3.1. The inverse lambda operators

This section presents the formal definition of the two components of the Inverse $\lambda$-Algorithms, $\text{Inverse}_L$ and $\text{Inverse}_R$. First, some definitions and explanations necessary to help understand the terminology used in defining $\text{Inverse}_L$ and $\text{Inverse}_R$ will be introduced. The objective of $\text{Inverse}_L$ and $\text{Inverse}_R$, is that given typed $\lambda$-calculus formulas $H$ and $G$, the formula $F$ is computed such that $F \circ G = H$ and $G \circ F = H$. Next, the different symbols used in the algorithm and their meanings are defined as follows:

- Let $G$, $H$ and $J$ represent typed $\lambda$-calculus formulas, $J^1, J^2, \ldots, J^n$ represent typed terms; $v$, $w$ and $v_1, \ldots, v_n$ represent variables

- Typed terms that are sub-terms of a typed term $J^i$ are denoted as $J^i_i$

We also consider the following two statements:

- A list of $\lambda$-abstractors of the form $\lambda v_1, \ldots, v_i$ can be empty if the corresponding variables $v_1, \ldots, v_i$ are not present in the formula they belong to.

- If the formulas being processed within the algorithm do not satisfy any of the if conditions then the algorithm returns null

**Definition 55** [operator : ] Consider two lists (of same length) of typed $\lambda$-calculus formulas $A_1, \ldots, A_n$ and $B_1, \ldots, B_n$, and a typed FOL $\lambda$-calculus formula $H$. The result of the operation $H(A_1, \ldots, A_n : B_1, \ldots, B_n)$ is defined as:

1. find the first occurrence of formulas $A_1, \ldots, A_n$ in $H$. 

49
2. replace each $A_i$ by the corresponding $B_i$.

3. find the next occurrence of formulas $A_1, ..., A_n$ in $H$ and go to 2. Otherwise, stop.

Before presenting the two algorithms, the intuition behind one of the expressions in the algorithm is introduced. With this example, it will become easier to read and understand the algorithms, although several examples will also be presented afterwards.

$$G \text{ is } \lambda v_1, ..., v_s. J^i(J^i_1, ..., J^i_m : v_{k_1}, ..., v_{k_m})$$

This expression indicates that the formula $G$ starts with a list of lambda abstractors and then it is followed by a typed term $J^i$, where some of its sub-terms $J^i_1$ to $J^i_m$, are substituted by variables $v_{k_p}$ where $1 \leq s \leq m$ and $\forall p, k_p \in \{1, ..., s\}$. For example, a valid expression for $G$ would be: $\lambda v_1.J(J_1, J_3 : v_1, v_1)$

Next, the definition of the two inverse algorithms is presented:

**Definition 56** $[Inverse_L(H, G)]$ The algorithm $Inverse_L(H, G)$, is defined as:

Given $G$ and $H$:

1. If $G$ is $\lambda v.v$
   
   $$F = \lambda v.(v@H)$$

2. If $G$ is a sub-term of $H$

   $$F = \lambda v.H(G : v)$$
3. $G$ is not $\lambda v.v$, $(J^1(J_{1}^{1},...,J_{m}^{1}), J^2(J_{1}^{2},...,J_{m}^{2}),..., J^n(J_{1}^{n},...,J_{m}^{n}))$ are sub-terms of $H$ and, $\forall J^i \in H$, $G$ is $\lambda v_1,...,v_s.J^i(J_{1}^{i1},...,J_{m}^{im})$ with $1 \leq s \leq m$ and $\forall p, 1 \leq k_p \leq s$.

- $F = \lambda w.H((J^1 : (w@J_{1}^{k_1} @...@J_{m}^{k_m})),...,J^n : (w@J_{1}^{k_1} @...@J_{m}^{k_m}))$ where each $J_{k_p}$ maps to a different $v_{k_p}$ in $G$.

4. $H$ is $\lambda v_1,...,v_i.J$ and $J^1(J_{i+1}^{1},...,J_{s}^{1})$ is a sub-term of $J$.

$G$ is $\lambda w.J^1(J_{i+1}^{1},...,J_{s}^{1}) : w@J_{1}^{k_1} @...@J_{m}^{k_s})$ with $\forall p, i+1 \leq k_p \leq s$.

- $F = \lambda w\lambda v_1,...,v_i.(w@\lambda v_{i+1},...,v_s.(J^1(J_{i+1}^{1},...,J_{s}^{1} : v_{k_1},...,v_{k_s})))$

Definition 57 [inverseR($H,G$)] The algorithm $inverseR(H,G)$, is defined as:

Given $G$ and $H$:

1. If $G$ is $\lambda v.v@J$

   - $F = inverseL(H,J)$

2. If $J$ is a sub-term of $H$ and $G$ is $\lambda v.H(J : v)$

   - $F = J$

3. $G$ is not $\lambda v.v@J$, $(J^1(J_{1}^{1},...,J_{m}^{1}), J^2(J_{1}^{2},...,J_{m}^{2}),..., J^n(J_{1}^{n},...,J_{m}^{n}))$ are sub-terms of $H$ and $G$ is $\lambda w.H((J^1(J_{1}^{1},...,J_{m}^{1}) : w@J_{1}^{k_1} @...@J_{m}^{k_s}),...,J^n(J_{1}^{n},...,J_{m}^{n}) : w@J_{1}^{k_1} @...@J_{m}^{k_s}))$ with $1 \leq s \leq m$ and $\forall p, 1 \leq k_p \leq m$.

\(^1\)When the formula $G$ is being generated, the indexes of the abstractors $\lambda v_1,...,v_s$ must be assigned to bind the variables from $v_{k_1},...,v_{k_m}$ in such a way that $G$ is a valid formula.
• \( F = \lambda v_1, ..., v_s.\mathcal{J}(J^1_{i+1}, ..., J^1_s : v_{k_1}, ..., v_{k_m}). \)

4. \( H \) is \( \lambda v_1, ..., v_i.\mathcal{J} \) and \( J^1(J^1_{i+1}, ..., J^1_s) \) is a sub-term of \( \mathcal{J} \),

\( G = \lambda w.\lambda v_1, ..., v_s.(w \mathcal{J}_{i+1}, ..., v_s.(J^1(J^1_{i+1}, ..., J^1_s : v_{k_1}, ..., v_{k_s}))) \) with \( \forall p, i+p \leq k_p \leq s. \)

• \( F = \lambda w.\mathcal{J}(J^1_{i+1}, ..., J^1_s) : w \mathcal{J}_{k_1} \mathcal{J}_{k_2} \mathcal{J}_{k_3} \)

3.1.1. Deterministic version of the Inverse Lambda Algorithms

Notice that the algorithm shown is non-deterministic. Depending on the sub-terms of \( H \) that one selects in option 3 and 4 of the algorithms, the output formula \( F \), satisfying \( H=F@G \) or \( H=G@F \), will be different. In order to make the algorithms deterministic, the concept of maximal sub-term will be introduced, which will guide the selection of sub-terms from the formulas. Its definition is the following:

**Definition 58** [Maximal sub-term] A sub-term \( M \) is a maximal sub-term of a typed \( \lambda \)-calculus formula \( F \) if there is no other sub-term \( N \), different from \( M \), such that \( M \) is a sub-term of \( N \) in \( F \).

Every time one has to select a sub-term from a formula, one will choose the maximal sub-term as long as it satisfies the conditions of the algorithm. In this way, one has a concrete process for selecting sub-terms and one will always obtain the same output formula \( F \). This step does not affect the proofs for soundness and completeness of the algorithms, since they are developed for any subsets of sub-terms that one may select from \( H \).

If one follows this behavior when selecting sub-terms, the \( Inverse_L \) algorithm can be reduced to three cases:
**Definition 59** [Deterministic $\text{Inverse}_L(H, G)$] The algorithm $\text{Inverse}_L(H, G)$, is defined as:

Given $G$ and $H$:

1. If $G$ is a sub-term of $H$
   
   - $F = \lambda v. H(G : v)$

2. $(J^1(J^1_1, \ldots, J^1_m), J^2(J^2_1, \ldots, J^2_m), \ldots, J^n(J^n_1, \ldots, J^n_m))$ are sub-terms of $H$ and, \( \forall J^i \in H, G \) is \( \lambda v_1, \ldots, v_s.J^i(J^i_1, \ldots, J^i_m : v_{k_1}, \ldots, v_{k_m})^2 \) with \( 1 \leq s \leq m \) and \( \forall p, 1 \leq k_p \leq s \).
   
   - $F = \lambda w.H((J^1_1 : (w@J^1_{k_1} @ \ldots @ J^1_{k_m})), \ldots, J^n : (w@J^n_{k_1} @ \ldots @ J^n_{k_m}))$ where each $J_{k_p}$ maps to a different $v_{k_p}$ in $G$.

3. $H$ is $\lambda v_1, \ldots, v_i.J$ and $J^1(J^1_{i+1}, \ldots, J^1_s)$ is a sub-term of $J$,
   
   - $G$ is $\lambda w.J(J^1(J^1_{i+1}, \ldots, J^1_s) : w@J^1_{k_1} @ \ldots @ J^1_{k_m})$ with $\forall p, i+1 \leq k_p \leq s$.
   
   - $F = \lambda w\lambda v_1, \ldots, v_i.(w@\lambda v_{i+1}, \ldots, v_s.(J^1(J^1_{i+1}, \ldots, J^1_s : v_{k_1}, \ldots, v_{k_s})))$)

In the deterministic version of $\text{Inverse}_L$, option 3 of the algorithm covers option 1 of the non-deterministic version. This is explained in the next example:

**Example 31:**

Let $H$ and $G$ be typed FOL $\lambda$-calculus formulas.

\[
H = \text{woman}(Mia) \land \text{man}(John)
\]

\[
G = \lambda x.x
\]

\(2G\) has to be a valid formula when the indexes of the abstractors $\lambda v_1, \ldots, v_s$ are assigned to bind the variables from $v_{k_1}, \ldots, v_{k_m}$.
These conditions satisfy option 1 of the non-deterministic version of the algorithm. The first step is to select sub-terms of H that satisfy the condition for the formula G. Following the deterministic process, one selects the maximal sub-term of H that satisfies the condition.

Consider \( J_1 = woman(Mia) \land man(John) \), since a typed term (in this case, formula) is a sub-term of itself. Consider \( J_1 = woman(Mia) \land man(John) \), for same reason as before.

Therefore, \( G = \lambda v_1, \ldots, v_s. J_i(J_1, \ldots, J_i : v_{k_1}, \ldots, v_{k_m}) \) which in this case \( G = \lambda v_1. J_i(J_1 : v_{k_1}) = \lambda v_1. J_i(\text{woman}(Mia) \land \text{man}(John) : v_1) = \lambda v_1. v_1 = G \).

Thus, \( F = \lambda w. H((J_1 : (w \@ J_1^1, \ldots, \@ J_1^{k_m}), \ldots, J_n : (w \@ J_n^1, \ldots, \@ J_n^{k_m}))) = \lambda w. H(J_1 : w \@ J_1^1) = \lambda w. H(\text{woman}(Mia) \land \text{man}(John) : w \@ \text{woman}(Mia) \land \text{man}(John)) = \lambda w. H(H : w \@ H) = \lambda w. w \@ H \). This is the output F of option 1 of the non-deterministic version of the algorithm.

### 3.2. Definition of the Inverse Lambda Algorithm and Complexity Analysis

In this section, the pseudocode of the Inverse \( \lambda \)-Algorithms is presented. Then, a study and analysis of the algorithms complexity is discussed. In the following subsection, the actual implementation of the algorithms is introduced along with an explanation of the different functions involved. First, the algorithm for \( \text{Inverse}_L \) presented in the previous section is described. It is presented in algorithm 1. After, \( \text{Inverse}_R \) is presented in algorithm 2. Both algorithms assume that G and H are typed lambda calculus formulas. J is a typed lambda calculus formula and F is initialized to a null value.

Next, the description of each of the functions presented in the algorithms will be introduced. In the next subsection, the implementation of the algorithm is discussed where
Algorithm 1 Algorithm for $\text{Inverse}_L$

```
BUILD TREE(H) ; BUILD TREE(G)
if EQUALS(G, $\lambda v. v$) then
    F = $\lambda v. (v @ + H + )$
else
    subtermsH = FINDSUBTERMS(H)
    for all subterm in subtermsH do
        if EQUALS(G, subterm) then
            F = $\lambda v + \text{SUBSTITUTION}(H, G, v)$
        end if
    end for
    if F is null then
        for all subterm in subtermsH do
            subsubterms = FINDSUBTERMS(subterm)
            J = GENERATEFORMULA(G, subterm, subsubterms, variablesFromG)
            if J is not null then
                applicatorTerm = GENERATEAPPLICATORTERM(variable, subsubterms)
                ADD(listFormulas, J) ; ADD(listApplicators, applicatorTerm)
            end if
        end for
        if listFormulas is not empty then
            F = $\lambda w + \text{SUBSTITUTION}(H, listFormulas, listApplicators)$
        else
            if H starts with a list of abstractors abs and a formula $J$ then
                subtermsJ = FINDSUBTERMS(J)
                for all subterm in subtermsJ do
                    subsubterms = FINDSUBTERMS(subterm)
                    applicatorTerm = GENERATEAPPLICATORTERM(variable, subsubterms)
                    if CHECKFORMFORMULA(G, J, subterm, applicatorTerm) is true then
                        vars = GENERATEVARIABLES(applicatorTerm)
                        lambdaAbstractors = GENERATEABSTRACTORS(vars)
                        F = $\lambda w + abs + (w @ + lambdaAbstractors + \text{SUBSTITUTION}(subterm, subsubterms, vars)$
                    end if
                end for
            end if
        end if
    end if
end if
```
Algorithm 2 Algorithm for $Inverse_R$

BUILDTREE(H)
BUILDTREE(G)
if $EQUALS(G, \lambda v. v @ + J)$ then
  $F = INVERSE_L(H, J)$
else
  subterms$H = FINDSUBTERMS(H)$
  for all subterm in subterms$H$ do
    $J = \lambda v + SUBSTITUTE(H, subterm, v)$
    if $EQUALS(G, J)$ then
      $F = J$
    end if
  end for
if $F$ is null then
  for all subterm in subterms$H$ do
    subsubterms$ = FINDSUBTERMS(subterm)$
    (J, applicatorTerm) = SEARCHSUBTERM_IN_H(subsubterms$)$
    if $EQUALS(G, \lambda w + SUBSTITUTE(H, J, applicatorTerm))$ then
      vars = GENERATEA_VARIABLES(applicatorTerm)
      lambdaAbstractors = GENERATE_ABSTRACTORS(vars)
      $F = lambdaAbstractors + SUBSTITUTE(subterm, subsubterms, vars)$
    end if
  end for
if $F$ is null then
  if $H$ starts with a list of abstractors $abs$ and a formula $J$ then
    subterms$J = FINDSUBTERMS(J)$
    for all subterm in subterms$J$ do
      subsubterms$ = FINDSUBTERMS(subterm)$
      if $CHECKFORMFORMULA(G, subterm, subsubterms)$ is true then
        applicatorTerm = GENERATE_APPLICATOR_TERM(variable, subsubterms$)$
        $F = \lambda variable + SUBSTITUTE(J, subterm, applicatorTerm)$
      end if
    end for
  end if
end if
each of the functions are explained and their complexity is calculated. The definition of
the functions from the algorithms are the following:

- **BUILD TREE(H)** - This function computes the tree of a formula H. This tree corre-
sponds to the first part of the derivation tree algorithm.

- **SUBSTITUTION(H, A, B)** - This function follows the specification presented in
  section 3.1 where for each appearance of A in H, each element of A is substituted
  by the corresponding element from B.

- **FINDSUBTERMS(H)** - This function processes the tree of a formula and applies
  the rules of occurrences of a particular languages to find all possible subterms of a
  formula.

- **EQUALS(H, G)** - This function takes the trees of H and G and checks whether H
  $\alpha$-converts to G, following the definition presented in chapter 2 of this work. If H
  $\alpha$-converts to G, then both formulas can be considered equal.

- **GENERATEFORMULA(G, subterm, subsubterms, variablesFromG)** - This func-
tion, using the formulas given as parameters and the specification of the formula G
from case 3 of $Inverse_{L}$, from the algorithm definition presented in the previous
section, tries to find a formula. If a formula can not be found using the formulas
given as parameters and the specification, then it returns null.

- **GENERATEAPPLICATORTERM(subsubterms)** - This function takes the formulas
given as parameters and creates an applicator term with them.
• GENERATEVARIABLES(applicatorTerm) - This function looks at the formulas of the applicatorTerm and their indexes and creates a list of variables with the same indexes.

• GENERATEABSTRACTORS(vars) - This function receives a list of variables as input and generates a list of abstractors that will bind these variables in a formula.

• CHECKFORMFORMULA(formula, formulas...) - This function checks if the first formula given as input can be constructed following the specification given in case 4 of InverseL or InverseR (depending on algorithm) and the rest of formulas given as parameters. If G can be built then the function returns true, false otherwise.

• SEARCHSUBTERM_IN_H(subsubterms) - This function looks for the formulas (subsubterms) given as input in the formula H. If they are found, then it returns the subterm they belong to and the applicator term that they can form. These two elements are then used to look for the formula G.

• OPERATOR + : This operator is used to join \( \lambda \)-elements and form expressions.

3.2.1. Complexity Analysis and Implementation

Consider the following problem:

INVERSE LAMBDA CALCULATION

Instance: Two typed lambda calculus formulas H and G of order two or less

Question: Is there a typed lambda calculus formula F of order two or less such that H = F @ G or of order one or less such that H = G @ F.

This problem belongs to the class NP.
Proof. Consider a yes-instance \( \{H, G\} \) of this problem and a certificate \( F \). One can easily verify that \( H = G @ F \) or \( H = F @ G \) by simply applying \( G \) to \( F \) or \( F \) to \( G \) and comparing the result to \( H \). It is clear that this operation can be computed in polynomial time on the size of the input formulas.

This problem most probably does not belong to the class co-NP.

To show that most likely this problem does not belong to co-NP, one can argue that to show that no typed lambda calculus formula \( F \) exists such that \( H = F @ G \) or \( H = G @ F \), it would be necessary to check every possible formula \( F \). One would need to check that no formula \( F \) can satisfy \( H = F @ G \) or \( H = G @ F \). The Inverse Algorithms look at the structure of the formulas \( H \) and \( G \) and from there, they try to find \( F \). If \( H \) and \( G \) are relatively small, it can be checked in polynomial time that no \( F \) is possible, but in the general case where \( H \) and \( G \) can be of any length, checking that there is no possible formula \( F \) takes exponential time.

Next, the current implementation of the functions presented in the previous subsection is discussed. Their complexity will also be presented and which heuristics were used and could be used for each of the operations of the algorithm. First, consider the generation of the tree of a certain formula:

Consider as input, two typed lambda calculus formulas \( H \) and \( G \). The length of \( H \) is \( n \) and the length of \( G \) is \( m \).

- **BUILDTREE(H)**
  - Running time: \( O(n) \) where \( n \) represents the length of \( H \).
- Implementation: The implementation of this function follows the specification presented in the part 1 of the derivation tree algorithm.

- **FINDSUBTERMS(H)**
  - Running time: $O(n)$ where $n$ represents the length of H.
  - Implementation: This function reads the tree of the formula H from top to bottom and following the definition of occurrences of the language, extracts subterms.

- **SUBSTITUTION(H, A, B)**
  - Running time: $O(n)$ where $n$ represents the length of H.
  - Implementation: This function substitutes all elements in A by the corresponding elements from B, for each appearance of A in H. We may substitute the entire H but no more than that, thus the complexity is based on the length of the formula.

- **EQUALS(H, G)**
  - Running time: $O(n)$ where $n$ represents the length of the smaller formula.
  - Implementation: This function starts reading from top to bottom the trees of the two formulas. Each node of the tree is compared to the same node on the other tree. If both trees contain the same elements then the function returns true, false otherwise.

- **GENERATEABSTRACTORS(variablesOfG)**
- Running time: $O(m)$ where $m$ represents the length of $G$.

- Implementation: This function returns a list of lambda abstractors that will bind the variables from $G$.

- **GENERATEAPPLICATORTERM**(variable, subsubterms)

  - Running time: $O(n)$ where $n$ represents the length of $H$.

  - Implementation: This function returns an applicator term built using the variable and subsubterms provided as input.

- **GENERATEVARIABLES**(applicatorTerm)

  - Running time: $O(n)$ where $n$ represents the length of $H$.

  - Implementation: This function returns a list of variables whose index matches the indexes of the formulas of the applicatorTerm.

- **GENERATEFORMULA**(G, subterm, subsubterms, variablesFromG)

  - Running time: $O((n!/(n - k)!) (n + 2m))$ where $n$ represents the length of $H$, $m$ is the length of $G$ and $k$ is the number of variables in $G$. The factorial term represents $n$ Permutation $k$ (arrangements or partial permutation).

  - Implementation: This function is implemented as follows.

    ```java
    for all possible partial permutation of subsubterms $P$ do
        abstractors = GENERATEABSTRACTORS(variablesFromG)
        $J = abstractors + SUBSTITUTION(subterm, P, variablesFromG)$
        if $EQUALS(G, J)$ then
    ```
return J

end if

end for

The loop has complexity $O(n!(n-k)!)$ since it is considering all the possible partial permutations of subsubterms. \textsc{Generateabstractors} and \textsc{equals} have complexity $O(m)$. \textsc{substitution} has complexity $O(n)$. 

- \textsc{checkformula}(formula, formulas...)

  - Running time: $O(2n)$ where $n$ represents the length of $H$.

  - Implementation: In the case of \textit{Inverse}_L, it refers to case 4. This function takes the subterm of the formula $J$ and substitutes it by the applicator term. If $G$ and the resulting formula are equal then it returns true, false otherwise. Substitution operation is $O(n)$ while the equals operation is $O(n)$. In the case of \textit{Inverse}_R, the subsubterms of the subterm are substituted by variables in the formula and then it is checked that the resulting formula follows the specification of $G$ in the case 4 of the algorithm. Again substitution and equality operation.

- \textsc{searchsubterm_in_h}(subsubterms)

  - Running time: $O(mn + 2m)$ where $n$ represents the length of $H$ and $m$ the length of $G$.

  - Implementation: This function looks for applicator terms in $G$ and then finds
the corresponding subterm of H from where it came from. It builds the formula G from H using the formulas found in the previous step, and finally, checks if the formula constructed following the specification of the case 4 of $Inverse_R$ equals the formula G given.

Since the algorithms are nested IF expressions, the complexity will be computed as the sum of the different 4 cases of each algorithm. In the case of $Inverse_L$, all functions presented run in polynomial time except for the function GENERATEFORMULA, which has a factorial component $(n!/(n-k)!).$ In the worst case, $m$ may correspond to the number of variables $k$, present in G. But normally, in the examples that have been tested and in corpuses used to test the semantic learning system, mentioned in the introduction, the number of variables in G is not greater than 4. Therefore when $k$ is small, the complexity is $\approx O(n^k)$. Thus, the complexity of the function GENERATEFORMULA is normally $O(n^4)(n+2m)$, polynomial. And thus, the complexity of $Inverse_L$ is polynomial as well. If the formula G happens to have a large number of variables, then the complexity of $Inverse_L$ increases to being factorial in the number of variables of G. This is due to the number of possible arrangements of the variables of G. $Inverse_R$ has polynomial time complexity since all the functions involved in the computation of the algorithm run in polynomial time. Next, we formalize these statements in the following theorems:

**Theorem 1** ($Inverse_L$ complexity). The $Inverse_L$ Algorithm runs in polynomial time in the size of the input formula H and G, when the formula G has a small number of variables with respect to the length of the formula H. Otherwise, the algorithm runs in factorial time in the number of variables of G.
**Theorem 2** (*$\text{Inverse}_R$* complexity). The *$\text{Inverse}_R$* Algorithm runs in polynomial time in the size of the input formula $H$ and $G$. 
4. EXAMPLES AND USE CASES

This chapter is started by presenting some examples of application of the two Inverse λ-Algorithms and how the different steps of each algorithm are calculated to obtain the result. Examples will be illustrated for the two formalisms covering different cases of the algorithms. This will be a first introduction to the algorithms application before some use cases are introduced in the next section of this chapter.

4.1. Inverse Lambda Algorithm Examples

4.1.1. Example 1 - Typed FOL lambda calculus

Let \( H \) and \( G \) be typed FOL λ-calculus formulas.

\[
H = \text{woman}(\text{Mia})
\]

\[
G = \lambda x. x
\]

\( F \) needs to be calculated such that \( H = F @ G \). Case 1 of \( \text{Inverse}_L \) will be applied.

Then \( F = \lambda v. (v @ H) \) and in this case \( F = \lambda v. (v @ \text{woman}(\text{Mia})) \).

\[
F @ G = \lambda v. (v @ \text{woman}(\text{Mia})) @ \lambda x. x = (\lambda x. x @ \text{woman}(\text{Mia})) = \text{woman}(\text{Mia}) = H.
\]

4.1.2. Example 2 - Typed FOL lambda calculus

Let \( H \) and \( G \) be typed λ-calculus formulas.

\[
H = \lambda u. ((\text{man}(\text{Vincent}) \land u) \rightarrow \text{man}(\text{Vincent}))
\]

\[
G = \text{man}(\text{Vincent})
\]

\( F \) needs to be calculated such that \( H = F @ G \). Case 2 of \( \text{Inverse}_L \) will be applied.

\[
F = \lambda v. H(G : v)
\]

\[
F = \lambda v. H(\text{man}(\text{Vincent}) : v)
\]

\[
F = \lambda v. \lambda u. ((v \land u) \rightarrow v)
\]
4.1.3. Example 3 - Typed FOL lambda calculus

Let $H$ and $G$ be typed $\lambda$-calculus formulas.

$$H = \lambda u. (\text{woman}(Mia) \land \text{happy}(Mia) \land \text{man}(Vincent) \land \text{happy}(Vincent) \land \text{loves}(Mia, u))$$

$$G = \lambda v. \lambda w. (v \land \text{happy}(w))$$

$F$ needs to be calculated such that $H = F @ G$. Case 3 of $\text{Inverse}_L$ will be applied.

$G$ is not $\lambda v. v$. That satisfies first condition.

From $H$, one has the following formulas that are sub-terms:

$$J_1^1 = \text{woman}(Mia) \land \text{happy}(Mia)$$

$$J_1^1 = \text{woman}(Mia)$$

$$J_1^2 = \text{Mia (from happy(Mia))}$$

$$J_2^1 = \text{man}(Vincent) \land \text{happy}(Vincent)$$

$$J_2^1 = \text{man}(Vincent)$$

$$J_2^2 = \text{Vincent (from happy(Vincent))}$$

that satisfies second condition. The third condition is satisfied by, $\forall J^i \in H: G = \lambda v_1. \lambda v_2. J^i(J^i_1, J^i_2 : v_1, v_2)$ for $i = 1, 2$.

For example, for $J^1$, $G = \lambda v_1. \lambda v_2. J^1(\text{woman}(Mia), Mia : v_1, v_2) = \lambda v. \lambda w. (v \land \text{happy}(w))$.

Therefore one can now calculate $F$:

Thus, $F = \lambda w. H((J^1 : w@J^1_1@J^1_2), (J^2 : w@J^2_1@J^2_2))$. Then,

$$F = \lambda x. H((J^1 : x@\text{woman}(Mia)@Mia), (J^2 : x@\text{man}(Vincent)@Vincent))$$

$$F = \lambda x. \lambda u. (x@\text{woman}(Mia)@Mia \land x@\text{man}(Vincent)@Vincent \land \text{loves}(Mia, u))$$
4.1.4. Example 4 - Typed ASP lambda calculus

Let $H$ and $G$ be typed $\lambda$-calculus formulas.

$$H = \text{fly}(X) \leftarrow \text{bird}(X), \text{not}\neg\text{fly}(X)$$

$$G = \lambda v. (\text{fly}(v) \leftarrow \text{bird}(X), \text{not}\neg\text{fly}(v))$$

$F$ needs to be calculated such that $H = F \circ G$. Case 3 of $Inverse_L$ will be applied.

From $H$, one has the following formulas that are sub-terms:

1. $J^1 = \text{fly}(X) \leftarrow \text{bird}(X), \text{not}\neg\text{fly}(X)$
2. $J^1_1 = X$
3. $J^1_2 = X$ (from second fly)

and that satisfies the second part of the condition. Then, the third condition is satisfied by:

$$G = \lambda v_1. J^1(J^1_1, J^1_2 : v_{k1}, v_{k2})$$

$$G = \lambda v_1. J^1(J^1_1, J^1_2 : v_1, v_1)$$

Therefore one can now calculate $F$:

$$F = \lambda w. H((J^1 : (w@J^1_{k1}, ..., @J^1_{km})), ..., J^n : (w@J^n_{k1}, ..., @J^n_{km}))$$

$$F = \lambda x. H((J^1(X, X) : x@X))$$

since $G$ has two equal $v_p$, only one $J_{kp}$ is taken.

$$F = \lambda x. x@X$$

4.1.5. Example 5 - Typed ASP lambda calculus

Let $H$ and $G$ be typed $\lambda$-calculus formulas.

$$H = \lambda u. (\text{happy}(Mia) \leftarrow \text{lovesBefore}(Mia, Vincent, u), \text{not}\neg\text{happy}(Mia))$$

$$G = \lambda v. (\text{happy}(Mia) \leftarrow v@Vincent@Mia, \text{not}\neg\text{happy}(Mia))$$

$F$ needs to be calculated such that $H = F \circ G$. Case 4 of $Inverse_L$ will be applied.

$$H = \lambda v_1. J$$

$$J = \text{happy}(Mia) \leftarrow \text{lovesBefore}(Mia, Vincent, u), \text{not}\neg\text{happy}(Mia)$$
\[ J^1(J^1_{i+1}, \ldots, J^1_s) = \text{lovesBefore}(Mia, Vincent, u) \text{ where: } J^1_2 = \text{Mia} \text{ and } J^1_3 = \text{Vincent} \]

\[ G \text{ is } \lambda v.J(J^1_{i+1}, \ldots, J^1_s) : v@J^1_{k_1} @ \ldots @ J^1_s = \]

\[ G = \lambda v.J(J^1_{i+1}, J^1_2 : v@J^1_{k_1} @ J^1_{k_2}) \]

\[ G = \lambda v.J(J^1(\text{Mia}, \text{Vincent}) : v@J^1_3 @ J^1_2) \]

\[ G = \lambda v.J(J^1(\text{Mia}, \text{Vincent}) : v@V\text{vincent}@\text{Mia}) \]

\[ F = \lambda w\lambda v_1, \ldots, v_s. (w@\lambda v_{i+1}, \ldots, v_s.J^1(J^1_{i+1}, \ldots, J^1_s : v_{k_1}, \ldots, v_{k_s})) \]

\[ F = \lambda w.\lambda u. (w@\lambda v_2, \lambda v_3. J^1(J^1_{i+1}, J^1_3 : v_{k_1}, v_{k_2})) \]

\[ F = \lambda w.\lambda u. (w@\lambda v_2, \lambda v_3. \text{lovesBefore}(v_3, v_2, u)) \]

4.1.6. Example 6 - Typed FOL lambda calculus

Let \( H \) and \( G \) be typed \( \lambda \)-calculus formulas.

\[ H = \exists x.(\text{plane}(x) \land \text{takes}(\text{John}, x)) \]

\[ G = \lambda v.(\exists x(v)) \]

\( F \) needs to be calculated such that \( H = G @ F \). Case 2 of \( \text{Inverse}_R \) will be applied.

From \( H \), the expression \( \text{plane}(x) \land \text{takes}(\text{John}, x) \) is a formula. Assign it the name \( J \).

\( G \) satisfies the second part of the condition, \( G \) is \( \lambda v.H(J : v) \), since in this case:

\[ G = \lambda v.H(\text{plane}(x) \land \text{takes}(\text{John}, x) : v) \]

Therefore, one can now calculate \( F \):

\[ F = J = \text{plane}(x) \land \text{takes}(\text{John}, x) \]

\[ G @ F = \lambda v.(\exists x(v)) @ \text{plane}(x) \land \text{takes}(\text{John}, x) = H. \]

4.1.7. Example 7 - Typed FOL lambda calculus

Let \( H \) and \( G \) be typed \( \lambda \)-calculus formulas.
\(H = \text{loves}(\text{Mia},\text{Vincent}) \wedge \text{loves}(\text{Mia},\text{Robert})\)

\(G = \lambda v.(v@\text{Mia}@\text{Vincent} \wedge v@\text{Mia}@\text{Robert})\)

\(F\) needs to be calculated such that \(H = G @ F\). Case 3 of \(\text{Inverse}_R\) will be applied.

\(G\) is not \(\lambda v.v@J\). In this case, \(J = \text{Mia}@\text{Vincent} \wedge v@\text{Mia}@\text{Robert}\) cannot be a formula using the definition and the algorithm for derivation trees. That satisfies the first condition.

From \(H\), one has the following formulas that are sub-terms:

\(J^1 = \text{loves}(\text{Mia},\text{Vincent})\)

\(J^1_1 = \text{Mia}\)

\(J^1_2 = \text{Vincent}\)

\(J^2 = \text{loves}(\text{Mia},\text{Robert})\)

\(J^2_1 = \text{Mia}\)

\(J^2_2 = \text{Robert}\)

and that satisfies the second part of the condition:

\(G = \lambda x.\text{H}((J^1(J^1_1, J^1_2) : x@J^1_{k_1} @ J^1_{k_2}),
(J^2(J^2_1, J^2_2) : x@J^2_{k_1} @ J^2_{k_2})\)

which in this case corresponds to \(G = \lambda x.\text{H}((\text{loves(\text{Mia},\text{Vincent})}: x@\text{Mia}@\text{Vincent}), (\text{loves(\text{Mia},\text{Robert})}: x@\text{Mia}@\text{Robert})).\)

Therefore, one can now calculate \(F\):

\(F = \lambda v_1, v_2.J^1(J^1_1, J^1_2 : v_1, v_2)\) and so \(F = \lambda v_1, v_2.\text{loves}(v_1, v_2)\)
4.1.8. Example 8 - Typed ASP lambda calculus

Let $H$ and $G$ be typed $\lambda$-calculus formulas.

\[ H = h(a) \text{ or } h(b) \rightarrow h(c, f), \neg h(d, f), \text{ not } \neg h(e). \]

\[ G = \lambda v. (h(a) \text{ or } h(b) \rightarrow v@c@f, \neg v@d@f, \text{ not } \neg h(e)). \]

$F$ needs to be calculated such that $H = G @ F$. Case 3 of $Inverse_R$ will be applied.

$G$ is not $\lambda v. v@J$, then first condition is satisfied.

From $H$, one has the following formulas that are sub-terms:

\[ J^1 = h(c, f) \]
\[ J^1_1 = c \]
\[ J^1_2 = f \]

\[ J^2 = h(d, f) \]
\[ J^2_1 = d \]
\[ J^2_2 = f \]

and that satisfies the second part of the condition:

\[ G = \lambda x. H((J^1(J^1_1, J^2_1) : x@J^1_k \oplus J^1_k), (J^2(J^1_2, J^2_2) : x@J^2_k \oplus J^2_k)) \]

\[ G = \lambda x. H((J^1(J^1_1, J^2_1) : x@J^1_k \oplus J^1_k), (J^2(J^1_2, J^2_2) : x@J^2_k \oplus J^2_k)) \]

Therefore one can now calculate $F$:

\[ F = \lambda v_1, ..., v_s. J^1(J^1_1, ..., J^1_m : v_{k1}, ..., v_{km}) \]

\[ F = \lambda v_1, v_2. J^1(J^1_1, J^1_2 : v_1, v_2) \]

\[ F = \lambda v_1, v_2. h(v_1, v_2) \]

4.1.9. Example 9 - Typed ASP lambda calculus

Let $H$ and $G$ be typed $\lambda$-calculus formulas.

\[ \text{F} \]
\[ H = \lambda v. (\text{stay}_at(\text{room}5) \leftarrow \text{not}_go_to_from(v, \text{room}5)) \]

\[ G = \lambda w. \lambda v. (w@\lambda u. \text{go}_to_from(v, u)) \]

\( F \) needs to be calculated such that \( H = G @ F \). Case 4 of \( \text{Inverse}_R \) will be applied.

\[ H = \lambda v. J \]

\[ J = \text{stay}_at(\text{room}5) \leftarrow \text{not}_go_to_from(v, \text{room}5) \]

\[ J^1(J^1, \ldots, J^1_s) = \text{go}_to_from(v, \text{room}5) \text{ with } J^1_2 = \text{room}5. \]

\( G \) is \( \lambda w. \lambda v_1. (w@\lambda v_2. (J^1(J^1_2 : v_{k_1}))) \)

\[ G = \lambda w. \lambda v_1. (w@\lambda v_2. \text{go}_to_from(\text{room}5 : v_2)) \]

Second condition satisfied. Thus:

\[ F = F = \lambda w. J^1(J^1_1 : w@J^1_2) \]

which in this case \( F = \lambda w. (\text{stay}_at(\text{room}5) \leftarrow \text{not}_w@\text{room}5). \)

**4.1.10. Example 10 - Typed FOL lambda calculus**

Let \( H \) and \( G \) be typed \( \lambda \)-calculus formulas.

\[ H = \lambda v. (\exists x. (\text{car}(x) \land \text{drives}(v, x))) \]

\[ G = \lambda w. \lambda v. (w@\lambda u. \text{drives}(v, u)) \]

\( F \) needs to be calculated such that \( H = G @ F \). Case 4 of \( \text{Inverse}_R \) will be applied.

\[ H = \lambda v. J \]

\[ J = \exists x. (\text{car}(x) \land \text{drives}(x, v)) \]

\[ J^1(J^1, \ldots, J^1_s) = \text{drives}(v, x) \text{ with } J^1_2 = x. \]

\( G \) is \( \lambda w. \lambda v_1. (w@\lambda v_2. (J^1(J^1_2 : v_{k_1}))) \)

\[ G = \lambda w. \lambda v_1. (w@\lambda v_2. \text{drives}(x : v_2)) \]

Second condition satisfied. Thus:
\[ F = F = \lambda w. J(J^1(\sigma_2) : \wedge_2 J^2) \]

which in this case \( F = \lambda w. \exists x. (\text{car}(x) \wedge w \wedge x) \).

4.1.11. Example 11 - Typed FOL lambda calculus

Let \( H \) and \( G \) be typed \( \lambda \)-calculus formulas.

\[ H = \lambda v. \lambda x. (\text{state}(x) \wedge v \wedge x) \]
\[ G = \lambda x. \text{state}(x) \]

\( F \) needs to be calculated such that \( H = F \circ G \). Case 3 of Inverse \( L \) will be applied.

From \( H \), one has the following formulas that are sub-terms:

\[ J^1 = \text{state}(x) \]
\[ J_1 = x \]

and that satisfies the second part of the condition:

\[ G = \lambda v_1. J^1(J_1 : v_1) \]

In this case: \( \lambda v_1. \text{state}(v_1) \)

Therefore one can now calculate \( F \):

\[ F = \lambda w. H((J^1 : (w \wedge J_1))) \]
\[ F = \lambda w. H((\text{state}(x) : w \wedge x)) \]
\[ F = \lambda w. \lambda v. \lambda x. (w \wedge x \wedge v \wedge x) \]

4.2. Use Cases

This section illustrates the usage of the algorithm to obtain the semantic representation of words from sentences. The first part introduces the concept of CCG parsing used to syntactically parse the example sentences. Then, the benefits that this method brings into the parsing process and semantic representation are discussed. For each of the formalisms presented in this work, a corresponding subsection will follow where the use of the al-
algorithm will be shown. Several sentences will be presented and unknown semantics of
words will be generated from the given lexicon and the meaning of the sentences.

4.2.1. Background: CCG parsing

The formalism that will be used to parse natural language sentences is called *Combinatory
Categorical Grammar* or *CCG*, [16]. Parses generated by this approach are built by the
combination of lexical elements that follow a simple set of rules. The main benefit of
this approach is that these rules concretely define which λ-expressions, associated with a
lexical element, should be applied to which lexical element. In this way, one has a defined
way in which the semantics of words from a sentence have to be combined to obtain the
complete representation of the sentence. A CCG parse specifies the logical form for each
sentence that can be parsed by the grammar.

Each lexical element consists of a word and a syntactic type or category and it may
also contain the semantic representation. The set of initial lexical elements is called *lexi-
con*. An example of a lexicon is the following:

\[
\begin{align*}
\text{Joe} & \rightarrow \text{NP} \\
\text{eats} & \rightarrow (S\setminus\text{NP})/\text{NP} \\
\text{pasta} & \rightarrow \text{NP}
\end{align*}
\]

In this lexicon, *Joe* and *pasta* are noun phrases and therefore they obtain the category
*NP*, which is a *basic category*. The word *eats* is a verb and it receives a more complex
category. It follows the role that a verb plays inside a sentence and its relation to the noun
phrases around it. The category \((S\setminus NP)/NP\) is formed by the basic categories \(S\) and
\(NP\) combined using the rules of \(A/B\) and \(A\setminus B\) combination, where \(A\) and \(B\) are basic
or complex types.
Now, two simple syntactic combination rules are formally presented. They can be used by CCG to construct trees:

- If one has a lexical element $x$ which category is $A$, and one has another lexical element $y$ which has the category $A/B$, then the application of the element $y$ to $x$, written as $yx$ is of category $B$.

- If one has a lexical element $x$ which category is $B$, and one has another lexical element $y$ which has the category $A \backslash B$, then the application of the element $x$ to $y$, written as $xy$ is of category $A$.

Intuitively, the category of $eats \ (S \backslash NP) / NP$ is representing that, if one has a noun phrase concatenated to the right of $eats$, one obtains a string of category $S \backslash NP$ which indicates that concatenating another noun phrase to the left, one obtains a string of category $S$ which is a sentence. In this case, if one concatenates $pasta$ to $eats$ one obtains $eats pasta$ which can be concatenated to $Joe$ to obtain $Joe eats pasta$ which is a sentence.

The CCG parse tree that represents this simple sentences looks like the following:

```
    Joe       eats       pasta
    NP       (S \ NP) / NP       NP
    NP       S \ NP
    S
```

Table 3

CCG parsing for “Joe eats pasta”.

The CCG lexicon can also contain semantic information for each lexical element. The previous lexicon can be extended in the following way:
In this lexicon, each lexical element has a syntactic type $A$ and a semantic type $f$. The two rules that were presented earlier can now be understood in the following way:

- If one has a lexical element $x$ which category is $A$ and semantic type is $f$, and one has another lexical element $y$ which has the category $A/B$ and the semantic type $g$, then the application of the element $y$ to $x$, written as $yx$ is of category $B$ and semantics $f(g)$.

- If one has a lexical element $x$ which category is $B$ and semantic type is $g$, and one has another lexical element $y$ which has the category $A\setminus B$ and semantic type is $f$, then the application of the element $x$ to $y$, written as $xy$ is of category $A$ and semantics $f(g)$.

When a combination rule is applied, the $\lambda$-calculus expression $f$ works as a function that receives as argument the $\lambda$-calculus expression $g$. The application returns a new $\lambda$-calculus expression of the combined category. The CCG parsing tree with semantic information is presented in table 4.

More combinatory rules can be applied to form more complex CCG trees and several extension to this formalism have been presented in the literature. Refer to [16] for more details on this topic and a full description of CCG parsing.
4.2.2. Increasing the lexicon of sentences in Typed FOL lambda calculus

The Use Cases examples begin by looking at sentences and their Typed First Order Logic Lambda Calculus representation from the database domain from [3].

Start with the sentence “Texas borders a state, what states border Texas?”. Then, consider a initial lexicon which includes the semantics for simple nouns and verbs. If one obtains the output of a simplified CCG parsing with two categories “S”, for sentences, and “NP”, for noun phrases, and one adds the semantics from the lexicon, the table 5 is obtained.

<table>
<thead>
<tr>
<th>Texas</th>
<th>borders</th>
<th>a</th>
<th>state</th>
<th>What</th>
<th>states</th>
<th>border</th>
<th>Texas</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP</td>
<td>(S\NP)/NP</td>
<td>NPNP</td>
<td>NP</td>
<td>S/(S\NP)NP</td>
<td>NPNP</td>
<td>S\NP</td>
<td>S\NP</td>
</tr>
<tr>
<td>NP</td>
<td>(S\NP)/NP</td>
<td>NPNP</td>
<td>NPNP</td>
<td>S/(S\NP)NP</td>
<td>S\NP</td>
<td>S\NP</td>
<td>S\NP</td>
</tr>
<tr>
<td>NP</td>
<td>(S\NP)</td>
<td>NPNP</td>
<td>S\NP</td>
<td>S\NP</td>
<td>S\NP</td>
<td>S\NP</td>
<td>S\NP</td>
</tr>
<tr>
<td>S</td>
<td>S</td>
<td>S</td>
<td>S</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Texas::
texas::
\[\lambda w.\lambda x. (w \circ \lambda y. \text{borders}(y,x))\]  
\[\lambda x. \text{state}(x)\]

What::

<table>
<thead>
<tr>
<th>What</th>
<th>border</th>
<th>Texas</th>
</tr>
</thead>
<tbody>
<tr>
<td>???</td>
<td>[\lambda x. \text{state}(x)]</td>
<td>???</td>
</tr>
<tr>
<td>???</td>
<td>[\lambda y. \lambda x. \text{borders}(x,y)]</td>
<td>???</td>
</tr>
<tr>
<td>???</td>
<td>[\lambda x. \text{state}(x)]</td>
<td>[\lambda x. \text{borders}(x,texas)]</td>
</tr>
<tr>
<td>???</td>
<td>[\lambda x. \text{state}(x)]</td>
<td>[\lambda x. \text{borders}(x,texas)]</td>
</tr>
</tbody>
</table>

Table 5

CCG and \(\lambda\)-calculus derivation for “Texas borders a state, what state borders Texas?”.
In table 5, one can see that the semantic representations for the words “a” and “What” are missing. These two words are not part of the initial lexicon but using the Inverse λ-Algorithms, their corresponding Typed First Order λ-Calculus representations can be computed. Starting with the first sentence, one can take the meaning of the sentence and the meaning of “Texas” to calculate the representation of “borders a state”.

One can see in the CCG tree that “borders a state” has category \((S \setminus NP)\) and therefore the λ-calculus expression will receive the word “Texas” from the right. Following this, if one takes H as the meaning of the sentence and G as the meaning of “Texas”, one can use \(Inverse_L(H, G)\) to obtain the expression for “borders a state”. In this case, option two of the algorithm is satisfied and \(F = \lambda y. \exists x. [state(x) \land borders(x, y)]\).

Now, one has the expression for “borders a state” and “borders”. One can calculate the expression of “a state” calling \(Inverse_R(H, G)\) with H being the meaning for “borders a state” and G being “borders”. Option four of the algorithm is satisfied and one obtains F as \(\lambda y. \exists x. [state(x) \land y@x]\). Finally, one calls \(Inverse_L(H, G)\) again with “What states” and “states” to obtain the desired meaning of “What”. Option three of the algorithm is satisfied and \(F = \lambda z. \lambda y. \lambda x. (z@x \land y@x)\). This is the Typed First Order λ-calculus representation for the simple word “a”.

The process to obtain the word “What” from the second sentence follows the same idea as just shown. First one calls \(Inverse_L(H, G)\) with H being the meaning of the sentence and G being “border Texas” to obtain the meaning of “What states”. Option three of the algorithm is satisfied and \(F = \lambda y. \lambda x. (state(x) \land y@x)\). Next, one calls \(Inverse_L(H, G)\) again with “What states” and “states” to obtain the desired meaning of “What”. Option three of the algorithm is satisfied and \(F = \lambda z. \lambda y. \lambda x. (z@x \land y@x)\).
Using the Inverse $\lambda$-Algorithms, the $\lambda$-calculus representation for the words “a” and “What” have been automatically added to the lexicon.

4.2.3. Increasing the lexicon of normative sentences and exceptions in Typed ASP lambda calculus

This Use Cases section continues by looking at normative sentences and exceptions and their Typed ASP Lambda Calculus representation from the examples presented in the paper [4]. These following sentences will be used:

- Most birds fly.
- Penguins are birds.
- Penguins do swim.
- Penguins do not fly.

As in the previous use case, one can consider a initial lexicon that has the semantics for simple nouns and verbs. After parsing the first two sentences above with CCG parsing and adding the semantics from the initial lexicon, one obtains the output of a simplified CCG parsing with two categories “S” and “NP”, as shown in table 6.

In table 6, one can see that the the semantic representations for the words “most” and “are” are missing. These two words were not part of the initial lexicon but one can compute their Typed ASP $\lambda$-calculus expression using the Inverse $\lambda$-Algorithms. Starting with the first sentence, one can take the meaning of the sentence and the meaning of the word “fly” to calculate the representation of “Most birds”.

78
Most birds fly

Penguins are birds

Table 6

CCG and λ-calculus derivation for “Most birds fly” and “Penguins are birds”.

“Most birds” has category NP and it is being applied from the right to the category of “fly”. Therefore, if one takes H as the meaning of the sentence and G as the meaning of “fly”, one can use InverseL(H, G) to obtain the expression for “Most birds”. In this case, option three of the algorithm is satisfied and F = λx.(x@X ← bird(X), not¬x@X).

Now, one has the expression for “Most birds” and “birds”. Since the word “bird” is being applied to the right of “Most birds”, one needs to again call InverseL(H, G) to obtain the representation for “Most”. Option three of the algorithm is again satisfied and one obtains F as λv.λx.(x@X ← v@X, not¬x@X). This is the Typed ASP λ-calculus representation for the word “most”.

The process to obtain the word “are” from the second sentence is very similar. First one calls InverseR(H, G) with H being the meaning of the sentence and G being “Penguins” to obtain the meaning of “are birds”. Option three of the algorithm is satisfied and F = λx.(x@X ← bird(X)). Next, one calls InverseL(H, G) with “are birds” and “birds” to obtain the desired meaning of “are”. Option three of the algorithm is satisfied again and
F = λu.λx.(x@X ← v@X). This expression corresponds to the Typed ASP λ-calculus formula for the word “are”. Next, the last two sentences are presented in table 7.

<table>
<thead>
<tr>
<th>Penguins</th>
<th>do</th>
<th>swim</th>
<th>Penguins</th>
<th>do not</th>
<th>fly</th>
</tr>
</thead>
<tbody>
<tr>
<td>NP (S/(S\NP)) \ NP S\NP</td>
<td>(S/(S\NP)) S\NP</td>
<td>S\NP</td>
<td>NP (S/(S\NP)) \ NP S\NP</td>
<td>(S/(S\NP)) S\NP</td>
<td>S\NP</td>
</tr>
<tr>
<td>Penguins</td>
<td>do</td>
<td>swim</td>
<td>Penguins</td>
<td>do not</td>
<td>fly</td>
</tr>
<tr>
<td>λx.penguin(x)</td>
<td>???</td>
<td>λx.swim(x)</td>
<td>λx.penguin(x)</td>
<td>???</td>
<td>λx.fly(x)</td>
</tr>
<tr>
<td>swim(X) ← penguin(X)</td>
<td></td>
<td></td>
<td>¬fly(X) ← penguin(X)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 7

CCG and λ-calculus derivation for “Penguins do swim” and “Penguins do not fly”.

In this case, the semantics for the words “do” and “do not” are missing. These two words were not part of the initial lexicon. As accomplished with the first two sentences, the meaning of the sentences and the known words will be used to obtain the needed missing semantics.

Starting with the first sentence and following the same reasoning shown before, one can call $Inverse_L$ to obtain the semantics for “Penguins do” (λx.(x@X ← penguin(X))). After this, one can call $inverse_R$ with the previous obtained semantic and the representation for “Penguins” to generate the desired meaning of “do” which corresponds to λu.λx.(x@X ← u@X). For the second sentence, one calls $Inverse_L$ first, obtaining λx.(¬x@X ← penguin(X)) and $Inverse_R$ afterwards, to obtain the representation for “do not” which is λu.λx.(¬x@X ← u@X).
5. THEOREMS AND PROOFS

**Theorem 3** (Soundness). Given two typed \( \lambda \)-calculus formulas \( H \) and \( G \) in \( \beta \)-normal form, if \( \text{Inverse}_L(H, G) \) returns a non-null value \( F \), then \( H = F \circ G \).

**Proof.** Let \( G \) and \( H \) be two typed \( \lambda \)-calculus formulas, let \( F = \text{Inverse}_L(H, G) \).

1. If \( G = \lambda v.v \)

   \[ F = \lambda v.(v \circ H) \]

   By the above condition of the algorithm for this case, \( H = H \) and \( G = \lambda v.v \). Then,
   \[ F \circ G = \lambda v.(v \circ H) \circ \lambda v.v = (\lambda v.v) \circ H = H. \]

2. If \( G \) is a sub-term of \( H \)

   \[ F = \lambda v.H(G : v) \]

   Let \( E_1 \) and \( E_2 \) be possibly empty series of \( \lambda \)-elements. By the above condition of the algorithm for this case, \( H = E_1 GE_2 \) and \( G = G \). Then, \( F \circ G = (\lambda v.E_1 E_2) \circ G = E_1 GE_2 = H \).

3. \( G \) is not \( \lambda v.v, (J^1(J_1^1, \ldots, J_m^1), J^2(J_1^2, \ldots, J_m^2), \ldots, J^n(J_1^n, \ldots, J_m^n)) \) are sub-terms of \( H \) and, \( \forall J^i \in H, G = \lambda v_1, \ldots, v_s.J^i(J_1^i, \ldots, J_m^i : v_{k_1}, \ldots, v_{k_m}) \) with \( 1 \leq s \leq m \) and \( \forall p, 1 \leq k_p \leq s \).

   \[ F = \lambda w.H((J^1 : (w \circ J_1^1 \circ \ldots \circ J_m^1), \ldots, J^n : (w \circ J_1^n \circ \ldots \circ J_m^n))) \]

   where each \( J_{kp} \) maps to a different \( v_{kp} \) in \( G \).

   Let \( E_1, E_2, \ldots, E_{n+1} \) be (possibly empty) series of \( \lambda \)-elements. By the above condition of the algorithm for this case, \( G = \lambda v_1, \ldots, v_s.J^i(J_1^i, \ldots, J_m^i : v_{k_1}, \ldots, v_{k_m}) \) and
\[ H = E_1 J^1 E_2 J^2 \ldots E_n J^n E_{n+1} \]. Then, \( F @ G = \)

\[
(\lambda w. E_1 w @ J^1_{k_1} @ \ldots @ J^1_{k_m} E_2 w @ J^2_{k_1} @ \ldots @ J^2_{k_m} \ldots E_n w @ J^n_{k_1} @ \ldots @ J^n_{k_m} E_{n+1}) @ G
\]

\[ G = E_1 (\lambda v_1, \ldots, v_s. J^i (J^1_1, \ldots, J^i_{m} : v_{k_1}, \ldots, v_{k_m})) @ J^1_{k_1} @ \ldots @ J^n_{k_m} E_2 \ldots E_n \]

\[
(\lambda v_1, \ldots, v_s. J^i (J^1_1, \ldots, J^i_{m} : v_{k_1}, \ldots, v_{k_m})) @ J^1_{k_1} @ \ldots @ J^n_{k_m} E_{n+1}
\]

\[ = E_1 J^1 (J^1_1, \ldots, J^i_{m}) E_2, \ldots, J^n (J^1_1, \ldots, J^i_{m}) E_{n+1} = H. \]

\( J^i \) contains all common \( \lambda \)-elements to all \( J^i \) formulas from \( H \), which sub-terms of each formula had been substituted by variables. Thus, when the formulas \( J^1_{k_1}, \ldots, @ J^i_{k_m} \) belonging to a given formula are placed in the variables \( v_{k_1}, \ldots, v_{k_m} \), one obtained the specific \( J^i \) formula back.

The reason why it is stated in \( G \) that each \( J_{k_p} \) maps to a different \( v_{k_p} \) is due to the last step of the proof. For each occurrence of a specific variable \( v_{k_p} \) in \( G \), one needs one formula \( J_{k_p} \) that will be placed on all the variables \( v_{k_p} \) during the application.

4. \( H \) is \( \lambda v_1, \ldots, v_i J \) and \( J^i (J^1_{i+1}, \ldots, J^1_s) \) is a sub-term of \( J \),

\( G \) is \( \lambda w. J (J^1_{i+1}, \ldots, J^1_s) : w @ J^1_{k_1} @ \ldots @ J^1_{k_s} \) with \( \forall p, i + 1 \leq k_p \leq s. \)

- \( F = \lambda w. \lambda v_1, \ldots, v_i (w @ \lambda v_{i+1}, \ldots, v_s. (J^1 (J^1_{i+1}, \ldots, J^1_s : v_{k_1}, \ldots, v_{k_s}))) \)

Let \( E_1, E_2 \) be (possibly empty) series of \( \lambda \)-elements. By the above condition of the algorithm for this case, \( H = \lambda v_1, \ldots, v_i. E_1 J^i (J^1_{i+1}, \ldots, J^1_s) E_2 \) and \( G = \lambda w. J (J^1_{i+1}, \ldots, J^1_s) : w @ J^1_{k_1} @ \ldots @ J^1_{k_s} \). Thus, \( F @ G = \)

\[
\lambda w. \lambda v_1, \ldots, v_i. (w @ \lambda v_{i+1}, \ldots, v_s. J^1 (J^1_{i+1}, \ldots, J^1_s : v_{k_1}, \ldots, v_{k_s})) @
\]

\[
\lambda w. J (J^1_{i+1}, \ldots, J^1_s) : w @ J^1_{k_1} @ \ldots @ J^1_{k_s} = \lambda v_1, \ldots, v_i. (\lambda w. J (J^1_{i+1}, \ldots, J^1_s)
\]

\[ : w @ J^1_{k_1} @ \ldots @ J^1_{k_s} @ \lambda v_{i+1}, \ldots, v_s. J^1 (J^1_{i+1}, \ldots, J^1_s : v_{k_1}, \ldots, v_{k_s})) \]

82
\[
\lambda v_1, \ldots, v_i. (J(\lambda v_{i+1}, \ldots, v_s. J^1_{i+1}, \ldots, J^1_s : v_{k_1}, \ldots, v_{k_s})) \@ J^1_{k_1} \@ \ldots \@ J^1_{k_s}) = \\
\lambda v_1, \ldots, v_i. J^1 (J^1_{i+1}, \ldots, J^1_s) = \lambda v_1, \ldots, v_i. E_1 (J^1 (J^1_{i+1}, \ldots, J^1_s)) E_2 = H.
\]

\[\square\]

**Theorem 4** (Soundness). Given two typed \(\lambda\)-calculus formulas \(H\) and \(G\) in \(\beta\)-normal form, if \(\text{Inverse}_R(H, G)\) returns a non-null value \(F\), then \(H = G @ F\).

**Proof.** Let \(G\) and \(H\) be two typed \(\lambda\)-calculus formulas, let \(F = \text{Inverse}_R(H, G)\).

1. If \(G\) is \(\lambda v. v @ J\)
   
   \[F = \text{Inverse}_L(H, J)\]

   One knows by the previous proof that if \(\text{Inverse}_L(H, J)\) returns a non-null value \(K\), then \(H = K @ J\). By the given condition of the algorithm for this case, \(G\) is \(\lambda v. v @ J\). Therefore, \(G @ F = \lambda v. v @ J @ K = K @ J = H\).

2. If \(J\) is a sub-term of \(H\) and \(G\) is \(\lambda v. H(J : v)\)
   
   \[F = J\]

   Let \(E_1\) and \(E_2\) be possibly empty series of \(\lambda\)-elements. By the above condition of the algorithm for this case, \(H = E_1 J E_2\) and \(G = \lambda v. E_1 (J : v) E_2\). Then, \(G @ F = (\lambda v. E_1 (J : v) E_2) @ J = E_1 J E_2 = H\).

3. \(G\) is not \(\lambda v. v @ J\), \((J^1 (J^1_{1, \ldots, J^1_m}), J^2 (J^2_{1, \ldots, J^2_m}), \ldots, J^n (J^n_{1, \ldots, J^n_m}))\) are sub-terms of \(H\) and \(G\) is \(\lambda w. H((J^1 (J^1_{1, \ldots, J^1_m}) : w @ J^1_{k_1} \@ \ldots \@ J^1_{k_m}), \ldots, (J^n (J^n_{1, \ldots, J^n_m}) : w @ J^n_{k_1} \@ \ldots \@ J^n_{k_m}))\) with \(1 \leq s \leq m\) and \(\forall p, 1 \leq k_p \leq m\).
\[ F = \lambda v_1, ..., v_s.J^1(J^1_{1_1}, ..., J^1_{m_1} : v_{k_1}, ..., v_{k_m}) . \]

Let \( E_1, E_2, ..., E_{n+1} \) be (possibly empty) series of \( \lambda \)-elements. By the above condition of the algorithm for this case, \( G = \lambda w.H((J^1_{1_1}, ..., J^1_{m_1}) : w @ J^1_{k_1} @ ... @ J^1_{k_m}) \) and \( H = E_1 J^1 E_2 J^2 ... E_n J^n E_{n+1} \). Thus, \( G @ F = (\lambda w.E_1 w @ J^1_{k_1} @ ... @ J^1_{k_m} E_2 \]

\[ w @ J^1_{k_1} @ ... @ J^1_{k_m} E_2 w @ J^1_{k_1} @ ... @ J^1_{k_m} E_{n+1} \) \( \@ \) \( (\lambda v_1, ..., v_s.J^1(J^1_{1_1}, ..., J^1_{m_1} : v_{k_1}, ..., v_{k_m}) = H \) as shown in the soundness proof of \textit{InverseL} since the common \( \lambda \)-elements to all \( J^i \) are also the common \( \lambda \)-elements to \( J^1 \).

4. \( H \) is \( \lambda v_1, ..., v_i.J \) and \( J^1(J^1_{i+1}, ..., J^1_s) \) is a sub-term of \( J \).

\( G \) is \( \lambda w.\lambda v_1, ..., v_i.(w @ \lambda v_{i+1}, ..., v_s.(J^1(J^1_{i+1}, ..., J^1_s : v_{k_1}, ..., v_{k_s}))) \) with \( \forall p, i+1 \leq k_p \leq s. \)

\[ F = \lambda w.J(J^1(J^1_{i+1}, ..., J^1_s) : w @ J^1_{k_1} @ ... @ J^1_{k_s}) \]

Let \( E_1, E_2 \) be (possibly empty) series of \( \lambda \)-elements. By the above condition of the algorithm for this case, \( H = \lambda v_1, ..., v_i.E_1 J^1(J^1_{i+1}, ..., J^1_s) E_2 \) and \( G = \lambda w.\lambda v_1, ..., v_i.(w @ \lambda v_{i+1}, ..., v_s.J^1(J^1_{i+1}, ..., J^1_s : v_{k_1}, ..., v_{k_s})) \). Thus, \( G @ F = (\lambda w.\lambda v_1, ..., v_i.(w @ \lambda v_{i+1}, ..., v_s.J^1(J^1_{i+1}, ..., J^1_s : v_{k_1}, ..., v_{k_s})) \) \( \@ \) \( \lambda w.J(J^1(J^1_{i+1}, ..., J^1_s) : w @ J^1_{k_1} @ ... @ J^1_{k_s}) = H \) as demonstrated in the soundness proof of \textit{InverseL}.

\[ \Box \]

**Definition 60** [applicator term] An applicator term of a typed \( \lambda \)-calculus formula is an expression of the form \( x @ J_1 @ ... @ J_s \) where \( x \) is a variable and \( J_i \) are formulas. The
different \( J_i \) need to be formulas due to the definition of typed \( \lambda \)-calculus formula.

**Definition 61** [sub-formula] A sub-formula of a typed \( \lambda \)-calculus formula \( F \) is a sub-term of \( F \) which is a typed \( \lambda \)-calculus formula.

**Theorem 5** (Completeness). For any two typed \( \lambda \)-calculus formulas \( H \) and \( G \) in \( \beta \)-normal form, where \( H \) is of order two or less, and \( G \) is of order one or less, if there exists a set of typed \( \lambda \)-calculus formulas \( \Theta_F \) of order two or less in \( \beta \)-normal form, such that \( \forall F_i \in \Theta_F, H = F_i \circ G \), then \( \text{Inverse}_L(H,G) \) will give an \( F \) where \( F \in \Theta_F \).

Given \( G \) and \( H \):

1. If \( G \) is \( \lambda v.v \)
   
   \[ F = \lambda v. (v \circ H) \]

2. If \( G \) is a sub-term of \( H \)
   
   \[ F = \lambda v. H( \circ v) \]

3. \( G \) is not \( \lambda v.v \), \( (J^1(J^1_1, ..., J^1_m), J^2(J^2_1, ..., J^2_m), ..., J^n(J^n_1, ..., J^n_m)) \) are sub-terms of \( H \) and, \( \forall J^i \in H, G \) is \( \lambda v_1, ..., v_s. J^i(J^i_1, ..., J^i_m : v_{k_1}, ..., v_{k_m}) \) with \( 1 \leq s \leq m \) and \( \forall p, 1 \leq k_p \leq s \).
   
   \[ F = \lambda w. H((J^1 : (w \circ J^1_{k_1} \circ ... \circ J^1_{k_m}), ..., J^n : (w \circ J^n_{k_1} \circ ... \circ J^n_{k_m}))) \] where each \( J^i_{k_p} \) maps to a different \( v_{k_p} \) in \( G \).

4. \( H \) is \( \lambda v_1, ..., v_i. J \) and \( J^1(J^1_{i+1}, ..., J^1_s) \) is a sub-term of \( J \),

   \( G \) is \( \lambda w. J(J^1(J^1_{i+1}, ..., J^1_s : w \circ J^1_{k_1} \circ ... \circ J^1_{k_s}), ..., J^n(J^n_{i+1}, ..., J^n_s : v_{k_1}, ..., v_{k_s}))) \) with \( \forall p, i+1 \leq k_p \leq s \).
   
   \[ F = \lambda w \lambda v_1, ..., v_i. (w \circ \lambda v_{i+1}, ..., v_s. (J^1(J^1_{i+1}, ..., J^1_s : v_{k_1}, ..., v_{k_s}))) \]

85
Proof. In section 2.5, it is demonstrated that there are 6 possible combinations of H, G and F. None of the combinations will satisfy option 4 of the algorithm since F needs to be order 3. The assumption from the condition of the theorem is that $H = F \ominus G$.

Proof by Contradiction.

- Case 1: Typed $\lambda$-calculus formulas H and G have order zero and F has order one.

1. Assume that $\text{Inverse}_L(H, G) = \text{null}$.

2. One has that option 1 and 3 of the algorithm are not satisfied since G is order zero.

3. From point 1 above, one knows that option 2 of the algorithm cannot satisfied.

4. From the previous point, one knows that a formula $G$ is not a sub-term of $H$.

5. Suppose $G$ is not a sub-term of $H$.

   – By definition of the input of the algorithm, G is order zero. By lemma 5, H is formed by $\lambda$-components of G and F. When G is placed in the outermost variable of F, it will not receive any modification since G is order zero. This outermost variable is a $\lambda$-term and by definition a sub-term. Therefore, $G$ is a sub-term of $H$. Contradiction.

- Case 2: Typed $\lambda$-calculus formulas H and F have order one and G has order zero.

1. Assume that $\text{Inverse}_L(H, G) = \text{null}$.

2. One has that option 1 and 3 of the algorithm are not satisfied since G is order zero.

3. From point 1 above, one knows that option 2 of the algorithm cannot satisfied.
4. From the previous point, one knows that a formula $G$ is not a sub-term of $H$.

5. Suppose $G$ is not a sub-term of $H$.

6. In case 1 above, it was shown that $G$ is a sub-term of $H$. Contradiction.

- Case 3: Typed $\lambda$-calculus formulas $H$ and $F$ have order two and $G$ has order zero.

1. Assume that $\text{Inverse}_L(H, G) = \text{null}$.

2. One has that option 1 and 3 of the algorithm are not satisfied since $G$ is order zero.

3. From point 1 above, one knows that option 2 of the algorithm cannot be satisfied.

4. From the previous point, one knows that a formula $G$ is not a sub-term of $H$.

5. Suppose $G$ is not a sub-term of $H$.

6. In case 1 above, it was shown that $G$ is a sub-term of $H$. Contradiction.

- Case 4: Typed $\lambda$-calculus formula $H$ has order zero, $G$ has order one and $F$ has order two.

1. Assume that $\text{Inverse}_L(H, G) = \text{null}$

2. Option 2 of the algorithm cannot be satisfied. Since $F$ has as input an order one formula, and a zero order as output, it needs to have an application to reduce the order. $H$ has zero order, therefore it does not have any application. Thus, $F$ cannot be formed by $H$ and be of second order.

3. From point 1 above, one knows that option 1 and 3 of the algorithm are not satisfied.
4. Option 1 of the algorithm is not satisfied if G is not of the form specified. One has two possible scenarios:

– If G is of the form considered in option 1, then it is satisfied and the algorithm returns the specified F. Contradiction.

– If G is not of the form considered in option 1, option 1 is not satisfied and one continues considering the last possible option.

5. From point 3 above, option 3 of the algorithm is not satisfied. For this to happen, some condition of option 3 cannot be satisfied. Thus, G is \( \lambda v.v \) and/or the formulas \( J^i \) are not sub-terms of H and/or \( \forall J^i \), G is not of the form \( \lambda v_1, \ldots, v_n J^i (J^i_1, \ldots, J^i_m : v_{k_1}, \ldots, v_{k_m}) \).

6. G is not \( \lambda v.v \); this was shown in point 4 above.

7. By definition of sub-term, a formula H has at least one sub-term which is H itself. Therefore, H has at least a sub-term \( J^0 \) which is itself.

8. G is order one, therefore, it will start with a list of lambda abstractors. By definition of Typed lambda-calculus formulas, after the list of the lambda abstractors, one will have a formula, call it J.

When G is applied to F, it is placed on the outermost variable. If there are more than one occurrences of this variable in F, one will have the formula G several times in the resulting formula H.

H, output of F, can be order one or zero, therefore each occurrence of the variable in F will be in an applicator term with the formulas found in H for each occurrence of G generating different versions of G in H. If H is order
two, then G will be order one, and in order to reduce the order of G to zero in 
F so that one obtains a valid formula H, one will also need applicator terms in 
F. Each of these versions of G in H can be identified as $J^i$.

These $J^i$ sub-terms can all have common $\lambda$-components depending on the 
structure of G. The sub-formulas of these $J^i$ are what differentiates one $J^i$ 
from another in H. Each of them has $m$ sub-formulas $J_i$ that will be placed 
in the variables of G. All $J^i$ have the same number of $J_i$ since they were 
all originated from the application of G to F, which has a definite number of 
variables. These sub-formulas belong to the applicators terms in F.

The variables in G that are bound to the initial list of abstractors can have any 
order or repetition inside G since there is no restriction on the structure of 
G. All stated to this point can be expressed as $G = \lambda v_1, \ldots, v_s. J^i(J^i_1, \ldots, J^i_m : v_{k_1}, \ldots, v_{k_m})$.

This is the second condition of option 3. Contradiction.

- Case 5: Typed $\lambda$-calculus formula H and G have order one, F has order two.

  1. Assume that $\text{Inverse}_L(H, G) = \text{null}$.

  2. Option 2 of the algorithm cannot be satisfied. Thus, G cannot be a sub-term 
of H. One has two possible scenarios:

    - Suppose F is a second order formula without applicator terms. When G 
is applied to F, G is placed in the outermost variable of F and it becomes 
a sub-term of H. Option 2 is satisfied. Contradiction.

    - Suppose F is a second order formula with applicator terms. When G is
applied to $F$, $\lambda$-abstractors and bound variables of $G$ are not present in $H$ when they are substituted by the formulas of the applicator terms of $F$.

Therefore, $G$ is not a sub-term of $H$ and option 2 is not satisfied. One proceeds considering other options of the algorithm.

3. By point 1 above, one also has that option 1 and 3 of the algorithm are not satisfied.

4. Option 1 of the algorithm is not satisfied if $G$ is not of the form specified. One has two possible scenarios:

   - If $G$ is of the form considered in option 1, then it is satisfied and the algorithm returns the specified $F$. Contradiction.
   
   - If $G$ is not of the form considered in option 1, option 1 is not satisfied and one continues considering the last possible option.

5. Option 3 cannot be satisfied. By point 4, one knows that $G$ is not a sub-term of $H$. By point 5, one knows that $G$ is not $\lambda v.v$. For option 3 to not be satisfied, some condition of option 3 cannot be satisfied. Thus, $G$ is $\lambda v.v$ and/or the formulas $J^i$ are not sub-terms of $H$ and/or $\forall J^i$, $G$ is not of the form $\lambda v_1, ..., v_s.J^i(J^1, ..., J^m : v_p, ..., v_q)$. If this is the case, then option 3 is not satisfied.

6. In this step, one can apply the same reasoning that was shown in the previous case (case 4), with the only difference being that in this situation, since $H$ is order one, the number of formulas in the applicators of $F$ will be at least one less than the number of initial $\lambda$-abstractors in $F$ or $G$.

90
Case 6: Typed $\lambda$-calculus formula $H$ and $F$ have order two, $G$ has order one.

1. Assume that $\text{Inverse}_L(H, G) = \text{null}$.

2. Option 2 of the algorithm cannot be satisfied. Thus, $G$ cannot be a sub-term of $H$. One has two possible scenarios:

   - Suppose $F$ is a second order formula without applicator terms. When $G$ is applied to $F$, $G$ is placed in the outermost variable of $F$ and it becomes a sub-term of $H$. Option 2 is satisfied. Contradiction.

   - Suppose $F$ is a second order formula with applicator terms. When $G$ is applied to $F$, $\lambda$-abstractors and bound variables of $G$ are not present in $H$ when they are substituted by the formulas of the applicator terms of $F$. Therefore, $G$ is not a sub-term of $H$ and option 2 is not satisfied. One proceeds considering other options of the algorithm.

3. By point 1 above, one also has that option 1 and 3 of the algorithm are not satisfied.

4. Option 1 of the algorithm is not satisfied if $G$ is not of the form specified. One has two possible scenarios:

   - If $G$ is of the form considered in option 1, then it is satisfied and the algorithm returns the specified $F$. Contradiction.

   - If $G$ is not of the form considered in option 1, option 1 is not satisfied and one continues considering the last possible option.

5. Option 3 cannot be satisfied. By point 4, one knows that $G$ is not a sub-term of $H$. By point 5, one knows that $G$ is not $\lambda v.v$. For option 3 to not
be satisfied, some condition of option 3 cannot be satisfied. Thus, G is $\lambda v.v$ and/or the formulas $J^i$ are not sub-terms of H and/or $\forall J^i$, G is not of the form $\lambda v_1, \ldots, v_s.J_i(J^i_1, \ldots, J^i_m : v_p, \ldots, v_q)$. If this is the case, then option 3 is not satisfied.

6. In this step, one can apply the same reasoning that was shown in case 4 above, with the difference being that in this situation, since H is order two, the result of applying the formula G to F will give a second order formula as output.

\[\square\]

**Theorem 6** (Completeness). For any two typed $\lambda$-calculus formulas H and G of order two or less in $\beta$-normal form, if there exists a set of typed $\lambda$-calculus formula $\Theta_F$ of order one or less in $\beta$-normal form, such that $\forall F_i \in \Theta_F$, $H = G \circ F_i$, then $Inverse_R(H, G)$ will give an F, where $F \in \Theta_F$.

The algorithm $Inverse_R(H, G)$, is defined as:

Given G and H:

1. If G is $\lambda v.v \circ J$
   - $F = Inverse_L(H, J)$

2. If J is a sub-term of H and G is $\lambda v.H(J : v)$
   - $F = J$

3. G is not $\lambda v.v \circ J$, $(J^1(J^1_1, \ldots, J^1_m), J^2(J^2_1, \ldots, J^2_m), \ldots, J^n(J^n_1, \ldots, J^n_m))$ are sub-terms of H and G is $\lambda w.H((J^1(J^1_1, \ldots, J^1_m) :$
$w \_ \_ J_{k_1}^{1} \_ \_ @ \_ \_ J_{k_m}^{1}, ..., (J_{i}^{1}(J_{i+1}^{1}, ..., J_{s}^{1}) : w \_ \_ @ \_ \_ J_{k_1}^{1} \_ \_ @ \_ \_ J_{k_m}^{1})$ with $1 \leq s \leq m$ and $\forall p, 1 \leq k_p \leq m$.

- $F = \lambda v_1, ..., v_s.J(J_{j_1}^{1}, ..., J_{j_m}^{1} : v_{k_1}, ..., v_{k_m})$.

4. $H$ is $\lambda v_1, ..., v_i.J$ and $J_{i+1}^{1}(J_{i+1}^{1}, ..., J_{s}^{1})$ is a sub-term of $J$,

$G$ is $\lambda w.\lambda v_1, ..., v_i.(w \_ \_ @ \_ \_ v_{i+1}, ..., v_s.(J(J_{i+1}^{1}, ..., J_{s}^{1}) : v_{k_1}, ..., v_{k_s})$) with $\forall p, i+1 \leq k_p \leq s$.

- $F = \lambda w.J(J_{i+1}^{1}, ..., J_{s}^{1} : w \_ \_ @ \_ \_ J_{k_1}^{1} \_ \_ @ \_ \_ J_{k_s}^{1})$

**Proof.** There are 6 possible combinations of $H, G$ and $F$ with different orders as discussed previously. None of the combinations will satisfy rule 4 of the algorithm since $G$ needs to be order 3. The assumption from the condition of the theorem is $H = G@F$.

Proof by Contradiction:

- Case 1: Typed $\lambda$-calculus formulas $H$ and $F$ have order zero and $G$ has order one.

1. Assume that $Inverse_R(H, G) = null$.

2. One has that option 1 and 3 of the algorithm are not satisfied since $G$ has to be order two to satisfy the conditions.

3. From point 1 above, one knows that option 2 of the algorithm is not satisfied.

4. This means that a formula $J$ is not a sub-term of $H$ or/and $G$ is not of the form $\lambda v.H(J : v)$.

5. Suppose that there is no sub-term $J$ of $H$.

-- By definition of sub-term, $H$ is a sub-term of $H$. Thus contradiction.
6. Suppose $G$ is not of the form $\lambda v. H(J : v)$.

---

- $G$ is a typed $\lambda$-calculus formula formed by $\lambda$-elements $g_1, g_2, \ldots, g_n$. $H$ is formed by $h_1, h_2, \ldots, h_n$. By point 5, $J$ is a sub-term of $H$, therefore $H$ is formed by $h_1, h_2, \ldots, h_i, j_1, \ldots, j_n, h_{i+1}, \ldots, h_n$ ($\lambda$-elements can be empty). By lemma 6, $G$ is $\lambda v. h_1, h_2, \ldots, h_i, v, h_{i+1}, \ldots, h_n$, which is in fact $\lambda v. H(J : v)$. Contradiction.

- Case 2: Typed $\lambda$-calculus formula $F$ has order zero and $H, G$ has order one.

1. Assume that $\text{Inverse}_R(H, G) = \text{null}$.

2. One has that option 1 and 3 of the algorithm are not satisfied since $G$ has to be order two to satisfy the conditions.

3. From point 1 above, one knows that option 2 of the algorithm is not satisfied.

4. This means that a formula $J$ is not a sub-term of $H$ or/and $G$ is not of the form $\lambda v. H(J : v)$.

5. In case 1 above, it was shown that $J$ is a sub-term of $H$ and that $G$ is of the form $\lambda v. H(J : v)$. Therefore, contradiction.

- Case 3: Typed $\lambda$-calculus formula $F$ has order zero and $H, G$ has order two.

1. Assume that $\text{Inverse}_R(H, G) = \text{null}$.

2. Option 1 is not satisfied since $F$ is order zero and it cannot be applied to $G$ applicator term.

3. Option 3 of the algorithm is not satisfied since $G$ needs to have applicator terms and $H$ has the same order as $G$. The outermost variable of $G$ could have
an occurrence not in an applicator term somewhere in the formula, but then this variable would have a different type.

4. From point 1 above, one knows that option 2 of the algorithm is not satisfied.

5. This means that a formula $J$ is not a sub-term of $H$ or/and $G$ is not of the form $\lambda v. H(J : v)$.

6. In case 1 above, it was shown that $J$ is a sub-term of $H$ and that $G$ is of the form $\lambda v. H(J : v)$. Therefore, contradiction.

- Case 4: Typed $\lambda$-calculus formula $H$ has order zero, $F$ has order one and $G$ has order two.

1. Assume that $Inverse_R(H, G) = \text{null}$.

2. Option 2 of the algorithm is not satisfied because $G$ is order two and $H$ is order zero. $G$ cannot be formed by $H$.

3. By point 1 above, option 1 and 3 of the algorithm are not satisfied.

4. Option 1 of the algorithm is not satisfied if $G$ is not of the form specified. One has two possible scenarios:

   - If $G$ is of the form considered in option 1, then it is satisfied and the algorithm returns the specified $F$. Contradiction.

   - If $G$ is not of the form considered in option 1, option 1 is not satisfied and one continues considering the last possible option.

5. By point 3 above, Option 3 of the algorithm is not satisfied. For this to happen, some condition of option 3 cannot be satisfied. Thus, $G$ is $\lambda v. v @ J$
and/or \( J^i \) are not sub-terms of \( H \) and/or \( G \) is not \( \lambda w.H((J^1(J_{i_1},...,J_{k_1}) : w@J_{p_1},..., @J_{q_1}),..., (J^n(J_{i_n},...,J_{k_n}) : w@J_{p_n},..., @J_{q_n})). \) If this is the case, then option 3 is not satisfied.

6. \( G \) cannot be \( \lambda v.v@J \); this was shown in the previous point 4.

7. By definition of sub-term, a formula \( H \) has at least one sub-term which is \( H \).

Therefore \( H \) has at least a sub-term \( J^0 \) which is itself.

8. \( G \) is a second order formula, therefore it is receiving a formula of order one or less as input. In order to do this, it needs an abstractor at the beginning of the formula that binds to the variable where the input formula will be placed. Denote the lambda abstractor by \( \lambda w. \).

\( H \), output of \( G \), can be order one or zero, therefore \( G \) will have occurrences of applicator terms whose variable will be bound to the initial abstractor to reduce the order. If \( H \) is order two, then \( F \) will be order one. In order to reduce the order of \( F \) to zero, so that one obtains a valid formula \( H \), one will also need applicator terms. The number of formulas in the applicator terms depends on the number of variables in \( G \). They will be equal if \( H \) is order zero and different if \( H \) is order one. Every occurrence of \( w \) will be in an applicator term for \( G \) to remain as a valid typed formula, since all occurrences of \( w \) need to have the same type as their corresponding abstractor.

By lemma 5, \( H \) is formed by \( \lambda \)-components of \( G \) and \( F \). Thus, in this case, all \( \lambda \)-components of \( G \) are present in \( H \). One has that the difference between the formula \( H \) and \( G \) is that \( H \) has as sub-terms, the formula \( F \) after being
applied to the applicator terms of G. F is order one, therefore, it will have a list of abstractors at the beginning of the formula which bind to variables in the formula. When F is applied to G, it will be placed in the variable $w$ of each applicator term and it will generate a formula when its variables are substituted by the formulas in the applicator term. These formulas will be part of H and each of them can be identified as $J^i$. Each of them will have common elements that belong to F, since F is only changing by the formulas of G placed on its variables.

All applicator terms will have the same number of formulas since they are all applied to the same F which has a specific number of variables. And since the variables in G can be in any order or repetition, the formulas of the applicators have to be as well.

All stated to this point can be expressed as: 

$$\lambda w.H((J^1(J_1, ..., J_m) : w@J_{k_1}, ..., @J_{k_m}), ..., (J^n(J_1, ..., J_m) : w@J_{k_1}, ..., @J_{k_m})).$$

This is the last condition of option 3. Contradiction.

• Case 5: Typed $\lambda$-calculus formulas F and H have order one, G has order two.

1. Assume that $\text{Inverse}_R(H, G) = \text{null}$.

2. Option 2 of the algorithm cannot be satisfied. One has two possible situations:
   - G is order two and H is order one. If G has no applicator terms, then G can be formed by H. And, as shown in case 1, this leads to the condition being satisfied and thus, contradiction.
   - If G has applicator terms, then it cannot be formed by an order one for-
mula H. Option 2 is not satisfied and one proceeds with the next possible options.

3. By point 1 above, option 1 and 3 of the algorithm are not satisfied.

4. Option 1 of the algorithm is not satisfied if G is not of the form specified. One knows by point 2 that G has at least one applicator term. One has two possible scenarios:

   – If G is of the form considered in option 1, then it is satisfied and the algorithm returns the specified F. Contradiction.

   – If G is not of the form considered in option 1, option 1 is not satisfied and one continues considering the last possible option.

5. Option 3 of the algorithm cannot be satisfied. For this to happen, some condition of option 3 cannot be satisfied. Thus, G is \( \lambda v.v@J \) and/or \( J^i \) are not sub-terms of \( H \) and/or G is not \( \lambda w.H((J^1(J_i,\ldots,J_k) : w@J_p,\ldots,N)(J^2(J_i,\ldots,J_k) : \ldots,\ldots,N)(J^n(J_i,\ldots,J_k) : x@J_p,\ldots,N)@J_q) \). If this is the case, then option 3 is not satisfied.

6. One can apply the same reasoning that was shown in the previous case (case 4) with the only difference being that in this case since H is order one, the number of formulas in the applicators of G needs to be at least one less than the number of initial \( \lambda \)-abstractors in F or G.

- Case 6: Typed \( \lambda \)-calculus formulas G and H have order two, F has order one.

   1. Assume that \( \text{Inverse}_R(H,G) = \text{null} \).
2. Option 1 of the algorithm cannot be satisfied. $G$ is receiving as input an order one formula $F$ to which a formula $J$ will be applied. The output of $G$ is $H$ of order two. In order for $J$ applied to $F$ to return a formula of order two is if $J$ was order two. but then, $G$ would become order three.

3. Option 2 of the algorithm cannot be satisfied. $G$ is receiving as input the formula $F$ of order one. In order to give a valid output of order two, $G$ needs to have some additional sub-term of order two and reduce the order of the input $F$ to zero so that the resulting expression is a formula. But, since $F$ is placed in a variable and not in an applicator term, its order cannot be reduced.

4. Option 3 cannot be satisfied. The reasoning for this option is the same as that in the previous case and it leads to a contradiction.
6. CLOSE RELATED WORK

6.1. Higher-order matching and Interpolation

The research presented in this work is similar to two problems called in the literature “higher-order matching” and “Interpolation problem”. Higher-order matching was first conjectured in [17] to be decidable. This problem consists of determining if a lambda calculus term, in the simply typed lambda calculus, is an instance of another. It can also be understood as solving the equation \( M = N \) where \( M \) and \( N \) are simply typed lambda terms and \( N \) is a closed term. More intuitively, the problem is to find a substitution \( \sigma \) assigning terms of consistent types to the free variables of \( M \) such that \( \sigma(M) \) admits \( N \) as its normal form. A proof that third-order matching is decidable is presented in [18].

In literature about higher-order matching, the order of atomic types is generally 1. This slightly differs from the definition presented in this work. Thus, third order in [18] has to be understood as second order in this research.

The Interpolation problem is presented in [18]. In the paper, it is used to show that higher-order matching is decidable by establishing relations between the two problems and their properties. The author divides the proof of decidability in two parts, the first considers a particular case which corresponds to the problem of interpolation and then the second reduces the general case to this particular case. The interpolation problem consists of a set of equations of the form \( (x \ c_1\ldots c_n) = b \) such that \( x \) is a free variable and \( c_1\ldots c_n \) and \( b \) are closed terms. A solution to this problem is a substitution \( \sigma \). A term \( t = \sigma x = \lambda y_1\ldots\lambda y_n.u \) and thus \( \sigma(x \ c_1\ldots c_n) = (\lambda y_1\ldots\lambda y_n.u \ c_1\ldots c_n) \). This term will reduce to \( u[c_1/y_1, \ldots, c_n/y_n] \) which in normal form is \( b \).

The higher-order matching and interpolation problems are defined with respect to an
extension of lambda calculus denoted as $\beta\eta$–calculus. $\eta$-calculus refers to the idea of
extensionality, which corresponds to considering two functions equal, if and only if, they
give the same result for any argument. Next, several new concepts related to this idea are
introduced.

**Definition 62** [$\eta$-redex] An $\eta$-redex is a $\lambda$-term $\lambda x.Mx$ with $x \notin \text{FV}(M)$. This term con-
tracts to $M$. $M$ is its contractum.

**Definition 63** [$\eta$-contraction] If a $\lambda$-term $P$ contains an $\eta$-redex occurrence $R$, and one
replaces that occurrence by its contractum, and the result is $Q$, then it is said that $P$ $\eta$-
contracts to $Q$.

**Definition 64** [$\beta\eta$-redex] An $\beta\eta$-redex is a $\beta$-redex or a $\eta$-redex.

**Definition 65** [$\beta\eta$-reduction] A $\lambda$-term $P$ $\beta\eta$-reduces to $Q$, written as $P \rightarrow_{\beta\eta} Q$, if and
only if $Q$ is obtained from $P$ by a finite (possibly empty) series of $\beta$-contractions or $\eta$-
contractions as well as necessary changes of bound variables.

The definition of $\beta\eta$-reduction given in [18] states this definition more simply:

**Definition 66** [$\beta\eta$-reduction [18]] The $\beta\eta$-reduction relation, written as $\triangleright_{\beta\eta}$, is defined
as the smallest transitive relation compatible with term structure such that

$$(\lambda x.tu) \triangleright_{\beta\eta} t[u/x]$$

$$(\lambda x.(tx)) \triangleright_{\beta\eta} t \text{ if } x \text{ is not free in } t$$

**Definition 67** [$\beta\eta$-normal form] A $\lambda$-term $P$ containing no $\beta\eta$-redexes is said to be in
$\beta\eta$-normal form.

The matching problem is defined in [18] as follows:
**Definition 68** [Matching Problem] A matching problem is a set \( \Phi = \{(a_1,b_1),\ldots,(a_n,b_n)\} \) of pairs of terms of the same type such that the terms \( b_1,\ldots,b_n \) are ground. A pair \((a,b)\) is frequently written as an equation \( a = b \).

**Definition 69** [Third Order Matching Problem] A third order matching problem is a matching problem \( \Phi = \{(a_1,b_1),\ldots,(a_n,b_n)\} \) such that the types of the variables that occur in \( a_1,\ldots,a_n \) are of order three at most.

**Definition 70** [Solution] Let \( \Phi = \{(a_1,b_1),\ldots,(a_n,b_n)\} \) be a matching problem. A substitution \( \sigma \) is a solution to this problem if and only if for every \( i \), the normal form of the terms \( \sigma a_i \alpha \)-converts to \( b_i \).

Another definition of the matching problem from [19] which has a more similar terminology to the one used in this work, is as follows:

**Definition 71** A matching problem is an equation \( v = u \) where \( v \) and \( u \) have type \( a \) and \( u \) is closed. A solution is a sequence of terms \( t_1,\ldots,t_n \) such that \( v[t_1/x_1,\ldots,t_n/x_n] \triangleright_\beta\eta u \) where \( v[t_1/x_1,\ldots,t_n/x_n] \) is the simultaneous substitution of \( t_i \) for each free occurrence of \( x_i \) in \( v \) for each \( i : 1 \leq i \leq n \).

In this article, the author states that it has been shown in the literature that a matching problem of certain order can be reduced to an interpolation problem of higher order. Therefore, the author continues the work considering an interpolation problem to show the decidability of the general matching problem at the end.

In the case of higher-order matching, if we consider only one term \( t_1 \) and the substitution of \( t_1 \) for the occurrences of \( x_1 \) in \( v \), then we have a problem of the form: \( (G \ t_1) \triangleright_\beta\eta H \) where \( G \) has occurrences of \( x_1 \). If we consider \( \beta \)-calculus instead and write \( t_1 \) as
$F$, then we obtain the problem of $Inverse_R$ presented in this work. Thus, one can see $Inverse_R$ as a special case of the higher-order matching problem considering $\beta$-calculus.

Similarly for the case of the interpolation problem introduced above, if we consider one term $c_1$ and the substitution $[c_1/y_1]$ in the term $u$ with $\beta$-calculus, one obtains the problem $(u\ c_1) \vdash_{\beta \eta} b$. It can be rewritten as $(F\ G) \vdash_{\beta \eta} H$, to use the same terminology previously explained in this research. Again, the same difference is that this work considers $\beta$-calculus.

Another important difference to mention between the definition of these two problems and the $Inverse$ problems presented in this work, is the form of the terms involved. In the case of $Inverse_L$ and $inverse_R$, terms $G,F$ and $H$ are typed lambda calculus formulas that, by definition, have been set as closed and $\lambda\text{-terms}$. Also, by definition of the Inverse $\lambda$-Algorithm, they are also considered in $\beta$-normal form. In higher-order matching and interpolation problems, the result of the substitutions is considered a closed term and in $\beta\eta$-normal form. But, the terms involved in the substitutions do not have the same restrictions that are set in the Inverse problems. Only in the case of interpolation, the terms $c_1, \ldots, c_n$ are considered closed, as well.

As stated above, in [19], it is discussed how the higher-order matching and interpolation problems are related, reducing the former to the latter. Considering $Inverse_R$ a special case of higher-order matching, and $Inverse_L$ as a special case of the interpolation problem, the work presented in this research also showed a relation between both $Inverse$ in the definition of the $Inverse_R$ algorithm. A $Inverse_R$ instance can be related to a $Inverse_L$ when the formula $G$ is of a specific form having $\lambda v. v@$ at the beginning of the formula.
All the discussion introduced to this point argues that the Inverse problems presented in this work can be considered as special cases of the higher-order matching and interpolation problems. As shown before, there are differences in terms of the theory of lambda calculus utilized, and in the considerations of the form of the different terms involved in the problems. In summary, Inverse$R$ is a special case of higher-order matching \(^1\) and Inverse$L$ is a special case of the interpolation problem.

However, the Inverse Algorithms provide an important contribution. As it is stated in [19], the higher-order matching problem, in the general case, is undecidable when using $\beta$-calculus, as shown in [20]. In this research, since Inverse$R$ is a special case of higher-order matching, an algorithm for computing a subset of the third order matching problem with $\beta$-calculus is presented. Also, the algorithm proposed in [18], where a set of solutions to a third order matching problem are enumerated, will not terminate if the problem admits no such set. In the case of Inverse$R$, the algorithm will simply return a null value and terminate.

6.2. Learning word-to-meaning mappings

In [21, 22], two algorithms are discussed that solve the problem of learning word-to-meaning mappings extended to the field of typed lambda calculus. In [21], the author presents an algorithm to infer lambda-terms based lexical semantics. This algorithm finds the minimal possible set of solutions of a mapping problem. The mapping problem defined in [21] is presented next, using the terminology previously introduced in this work.

Definition 72 [BCI-$\lambda$-term] A BCI-$\lambda$-term or linear term is $\lambda$-term $P$ such that:

\(^1\)In a recent communication with Gerard Huet, he also interpreted the Inverse$L$ problem as a special case of matching.
• for each sub-term $\lambda x. M$ of $P$, $x$ occurs free in $M$ exactly once,

• each free variable of $P$ has just one occurrence free in $P$.

**Definition 73** [Mapping Problem] Consider $m$ sets of distinct variables $\bar{x}_1, ..., \bar{x}_m$. Let $T$ be a $\lambda I$-term with free variables $\bar{x}_1, ..., \bar{x}_m$ in $\beta \eta$-normal form. Find $m$ $\lambda I$-terms $S_{\bar{x}_1}^{T_1}, ..., S_{\bar{x}_m}^{T_m}$ and a BCI$\lambda$-term $D$ (with free variables $y_{\bar{x}_1}^{T_1}, ..., y_{\bar{x}_m}^{T_m}$) in $\beta \eta$-normal form, such that

$$(...((D \ S_1) \ S_2)... \ S_m) \triangleright_\beta \eta \ T.$$  

In the mapping problem defined above, the solution $S_{\bar{x}_1}^{T_1}, ..., S_{\bar{x}_m}^{T_m}$ represents the meaning of words while $D$ represents the meaning of the sentence. This mapping problem uses $\beta \eta$-calculus and tries to find $m$ terms that applied to a term $D$ will give the term $T$ in normal form. One can see that it is very similar to the matching problem presented in the previous section. Therefore, the *Inverse* problem presented in this research is again slightly different than this approach.

If one considers only one term $S_1$ that needs to be applied to $D$ in order to obtain $T$, the problem becomes $(D \ S_1) \triangleright_\beta \eta \ T$, which corresponds to $Inverse_R$ if one uses $\beta$-calculus instead. In this case, the terms involved in this mapping problem are in normal form and $\lambda I$-terms, which is more similar to the *Inverse* definition than the higher-order matching problem discussed in the previous section. But again, the definition of the problem is not the same since the theory of lambda calculus is different and the *Inverse* problem considers closed terms.

In [22], the mapping problem is redefined in the following terms:

**Definition 74** [Mapping problem [22]] Given a closed $\lambda I$-term $N$ of type $t$ in $\beta$-normal form containing constant symbols $c_1, ..., c_m, d_1, ..., d_n$, find a closed $\lambda I$-term $M$ of type...
such that

- The constant symbols appearing in $M$ are $c_1, \ldots, c_m$.

- There is a $\lambda I$-term $P[z^E]$ of type $t$ with $z^E$ as its only free variable such that $P[M]$ $\beta$-reduces to $N$.

In this case, $N$ is the meaning of the sentence, $c_1, \ldots, c_m$ is the set of constants that belong to one of the words of the sentence, and $P[z^E]$ is obtained by combining the meaning of a sentence and the rest of words of that sentence. One can see that this definition of the mapping problem is very similar to the definition of $\textit{Inverse}$ presented in this work. It considers $\beta$-calculus and closed $\lambda I$-terms. This definition is closely related to the $\textit{Inverse}_R$ definition.

In [21], an algorithm to find a “reasonably” small set of solutions to the mapping problem introduced previously, is presented. The author comments that one of the problems with the algorithm is the compatibility between the meaning already found for words in previous executions, and the new meaning obtained in a current sentence-meaning pair. An upper bound needs to be established in order to handle this kind of situations and including this behavior in the overall algorithm was left for further research by the author. In the case of $\textit{Inverse}$, there is no incompatibility between meanings of words in the learning system, to which the $\textit{Inverse}$ algorithms belong, since it handles different meanings of words with associated probabilities from the training set.

Another key difference between this approach and the one presented in this work, refers to the assumptions taken before executing the algorithm. In [21, 22], there is a first phase where the learner, in some way, obtains the set of constants that each of the
words have. Then, this information is used in the second phase to obtain the correct representation of words based on the constants. In the Inverse problem, the unknown representation of words in a sentence is obtained using the meaning of the sentence and the meaning of known words. There is no previous information needed about the unknown meaning.
7. CONCLUSION

In this work, two Inverse $\lambda$-calculus Algorithms were presented and their application to typed First-Order Logic $\lambda$-calculus and typed Answer Set Programming lambda calculus has been shown. With this algorithm, it is possible to automatically obtain semantic representations of unknown words using the information already available from known sentences and words.

After an introduction of lambda calculus and several of its main basic definitions, it was presented how typed lambda calculus can be merged with other formalisms to create languages which present interesting properties. These languages are used with the Inverse $\lambda$-Algorithms and the relation between the original and the new formalisms was shown. One can start with a formula in typed FOL lambda calculus representing the meaning of a word and after several applications with other formulas, a first-order formula is obtained. This first-order formula actually represents the meaning of a sentence. This meaning was obtained from the composition of the meaning of words represented in typed FOL lambda calculus.

To easily understand the relation between formulas and types, an algorithm for building derivation trees for the different typed lambda calculus formulas of the distinct formalism was presented. Using this algorithm, one can infer the type of typed lambda calculus expressions in the different formalisms and decide whether an expression has been properly built or not. The type and rule number associated to each node of the tree provides useful information to syntactically analyze the expression. For each of the formalisms, types and derivation rules were defined and strictly stated how formulas are built and how they are applied to obtain other formulas through well-typed applications.
For the two algorithms presented, $Inverse_L$ and $Inverse_R$, examples with the different formalisms were presented and a section with use cases for each of the languages showing real examples of the use of the algorithm. A deterministic version of the algorithm was also defined where the concept of maximal sub-term drives the choice of sub-terms of formulas.

For the Inverse $\lambda$-Algorithms, a completeness proof up to second order expressions has been developed. Blackburn and Bos [2] mention that natural language semantics rarely requires types above order three. The future work for this research is to create completeness results for third order expressions. This type of expressions are present in the last option of each of the algorithms. In the option 4 of $Inverse_L$, $F$ is a third order formula, and in the case of $Inverse_R$, $G$ is a third order formula.

As it was stated earlier in this work, the Inverse $\lambda$-Algorithms can be used with any knowledge representation language. During the development of this research, the languages of Propositional Linear Temporal Logic [23, 24] and Concurrent Propositional Dynamic Logic [25, 26] were merged with typed lambda calculus to be used by the algorithms. This step helped proving that apart from first-order logic and answer set programming, the algorithms could be applied to other KR languages. These languages are not included in this work so that they could be further explored and studied. They have been set as future work of this research.

During this research, it was realized that some words, for example verbs, would have two different semantics depending on the syntactic category associated. Normally this will depend on the sentence construction or on the combination of the verb with other words in the sentence. If a sentence is built from the category of a verb, then the verb
will usually have a fairly simple $\lambda$-calculus expression of second order. Otherwise, the meaning of the verb can be combined with the sentence when the structure is already formed by other phrases. In this case, the representation of the verb would be much more complex and of higher order. This is one of the reasons why extending these algorithms to third order completeness is a next immediate step to this research. By doing this, the algorithms will be able to compute the meanings of words in situations where one has complex representations of verbs.

These two algorithms are an important part of a natural language learning system which learns the semantic representations of words from sentences. The algorithms were implemented as part of that system. The results obtained by this system are quite promising. It outperforms earlier systems with respect to the F-measure for the two standard corpus GEOQUERY and CLANG.
REFERENCES


