The Inverse Lambda Calculus Algorithm for Typed First Order Logic Lambda Calculus and Its Application to Translating English to FOL

Chitta Baral[1], Marcos Alvarez Gonzalez[1], and Aaron Gottesman[1]

School of Computing, Informatics, and Decision Systems Engineering
Arizona State University, Tempe, AZ

Abstract. In order to answer questions and solve problems that require deeper reasoning with respect to a given text, it is necessary to automatically translate English sentences to formulas in an appropriate knowledge representation language. This paper focuses on a method to translate sentences to First-Order Logic (FOL). Our approach is inspired by Montague's use of lambda calculus formulas to represent the meanings of words and phrases. Since our target language is FOL, the meanings of words and phrases are represented as FOL-lambda formulas. In this paper we present algorithms that allow one to construct FOL-lambda formulas in an inverse manner. Given a sentence and its meaning and knowing the meaning of several words in the sentence our algorithm can be used to obtain the meaning of the other words in that sentence. In particular the two algorithms take as input two FOL-lambda formulas G and H and compute a FOL-lambda formula F such that F with input G, denoted by $F@G$, is H; respectively, $G@F = H$. We then illustrate our algorithm and present soundness, completeness and complexity results, and briefly mention the use of our algorithm in a NL Semantics system that translates sentences from English to formulas in formal languages.

1 Introduction

Our overall goal is to translate English to formal logics and Knowledge Representation (KR) languages. In this, our approach is to use λ-calculus formulas [1] to represent the meaning of words, phrases and sentences as previously done in [2], [3], [4] and [5]. To construct the meaning of a sentence from the meaning of its constituent words and phrases we use Combinatory Categorial Grammar (CCG) [6]. A big challenge in the above approach is to come up with the λ-calculus formulas that represent the meaning of the words. In some of the above mentioned works, they were either constructed by the authors or generated by a hand crafted set of rules. However, such an approach is not scalable. By analyzing how a human would come up with the λ-calculus formulas of words we noticed that at times a human, given G and H, would try to construct a formula F such that $F@G = H$ or $G@F = H$.

We illustrate the basic idea of inverse application of FOL-λ-calculus formulas and how they can be used in constructing the FOL-λ-calculus expressions of words through the following example. Here we take an example from the database querying domain of [7] that is presented in Table 1.

$$\begin{array}{cccc}
\text{Texas} & \text{borders} & \text{a} & \text{state} \\
NP & (S\backslash NP)/NP & NP/NP & NP \\ \cline{3-4}
NP & (S\backslash NP)/NP & \multicolumn{2}{c}{NP} \\ \cline{2-4}
NP & \multicolumn{3}{c}{(S\backslash NP)} \\ \hline
\multicolumn{4}{c}{S}
\end{array}$$

$$\begin{array}{cccc}
\text{Texas} & \text{borders} & \text{a} & \text{state} \\
texas & \lambda w.\lambda x.(w@\lambda y.borders(y,x)) & ??? & \lambda x.state(x) \\ \cline{3-4}
texas & \lambda w.\lambda x.(w@\lambda y.borders(y,x)) & \multicolumn{2}{c}{???} \\ \cline{2-4}
texas & \multicolumn{3}{c}{???} \\ \hline
\multicolumn{4}{c}{\exists x.[state(x) \wedge borders(x, texas)]}
\end{array}$$

Table 1. CCG and λ-calculus derivations for "Texas borders a state."

In Table 1 it is assumed that one knows the meaning or translation of the sentence "Texas borders a state" and knows the FOL-λ-calculus formulas for "Texas", "borders", and "state". But the semantic representation for the word "a" is not known. The first step to compute this missing semantic representation is to take the meaning of the sentence "Texas borders a state" and the meaning of the word "Texas" to compute the semantic representation of "borders a state". But to do that one needs to know or a-priori decide whether it is appropriate to use the meaning of "Texas" as an input to the meaning of "borders a state" or vice versa. Traditional grammars such as Context Free Grammars do not help us in this. On the other hand, Combinatory Categorial Grammars (CCGs) [8] provide us directionality information that can be used in deciding which is the input and which is the function.

In Table 1 the top part gives a CCG parse of the sentence "Texas borders a state". The various symbols correspond to basic categories; "S" represents a sentence, "NP" represents a noun phrase; and more complex categories are formed using "\" and "/" slashes which specify directionality. For instance, the category "$S\backslash NP$" specifies that when a phrase corresponding to that category is concatenated on the left with a phrase of category "NP" , then the resulting phrase is of the category "S"; i.e., a sentence. Intuitively, a non-transitive verb has a category "$S\backslash NP$", meaning that if a noun phrase is added on the left, we get a sentence. Similarly, the category "NP/NP" means that a phrase of that category concatenated on the right by a phrase of category "NP" results in a phrase of the category "NP".

The phrase "Texas" has a category "NP" and it is being applied from the right to "borders a state". Therefore, if we take H as the meaning of the sentence and G as the meaning of "Texas", we can see by inspection that an F, the meaning of "borders a state", such that $F@G = H$ can be $F = \lambda y.\exists x.[state(x) \wedge borders(x, y)]$.

Now we can take H as the meaning of "borders a state" and G as the meaning of "borders", which has category $(S\backslash NP)/NP$, and we can can find the meaning of F, for "a state". Since "a state" has category NP and is applied to the right of "borders", we need to find F such that $G@F = H$ which can be $F = \lambda y.\exists x.[state(x) \wedge y@x]$. Finally, with H being the expression for "a state" and G being the expression for "state", with category NP, we can obtain F, a representation for "a", with category NP/NP. Therefore, we need $F@G = H$, which can be $F = \lambda z.\lambda y.\exists x.[z@x \wedge y@x]$. This is the typed first-order λ-calculus representation for the simple word "a". But how exactly did we compute it?

The main goal of this paper is to address that and present algorithms which are capable of automatically computing F such that $F@G = H$ or $G@F = H$. It is this task we refer to as the *Inverse λ-Problem* and our algorithms as *Inverse λ-Algorithms*. The algorithm that computes F such that $F@G = H$, we call $Inverse_L$, while the second algorithm which computes F such that $G@F = H$, we call $Inverse_R$. Although our algorithms are defined in a way that they can be useful with respect to multiple KR languages, in this paper we focus on First-Order Logic as the KR language.

To help in showing the correctness and applicability of our $Inverse_L$ and $Inverse_R$ algorithms we need to recap the notion of typed λ first order theories and the notion of order in such theories, as these notions will be used when we present the soundness, completeness and complexity results of our Inverse λ-Algorithms. For example, the completeness result is with respect to typed first order theories up to the second order. This recapping is done in Section 2.

The rest of this paper is organized as follows. In Section 3 we present the Inverse λ-Algorithms. In Section 4 we illustrate our algorithms with respect to several examples. In Section 5 we show a use case example. In section 6 we present the complexity, soundness and completeness results. In Section 7 we discuss related work and other approaches in solving the Inverse λ-Problem. Finally, in Section 8 we conclude and briefly mention the companion learning based natural language semantics system that uses our algorithms as described in [9].

Overall, the main contributions of this paper are the development of the Inverse λ-Algorithms for typed λ first order theories and the presentation of soundness, completeness and complexity results for these algorithms.

2 Background

Montague, [10], was the first to suggest that natural languages need not be treated differently from formal languages and they could have formal semantics. The approach of using λ-calculus to represent the meaning of words and λ-application as a mechanism to construct the meaning of phrases and sentences is also considered to originate from Montague. As mentioned before, that is also our approach.

However, to further ground the notion of "meaning" (or semantics) it is useful to have a notion of models of λ-calculus expressions. Such notions also allow us to evaluate our natural language meaning representations. Both untyped and typed λ-calculus can be characterized using models, but the one that has had the most impact on natural language semantics is typed λ-calculus, as creating models for typed λ-calculus theories is somewhat simple. When we refer to model, we are looking for a semantic tool that can give us two elements: the entities that we have in our domain, and for every element in our signature, the semantic value associated with it.

We will use the "Simply Typed Lambda Calculus" of Church (1940), since it is most commonly used in linguistics. In this theory, there is only one type constructor, $\to$, to build function types and each term has a single type associated with it [11]. The books [12] and [13] are good reference points for typed lambda calculus.

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 we focus our Inverse λ-Algorithms on it, particularly on Typed First-Order Logic Lambda Calculus. For obtaining formal results, we need to define the signature of our language, the construction of typed terms and typed formulas, and the notion of sub-terms of a typed term in the language.

We start by introducing the signature of the Typed First-Order Logic Lambda Calculus Language (Typed FOL Lambda Calculus). It consists of:

- The lambda operator (also called the lambda-abstractor) λ, the lambda application @, the parenthesis $(,), [,]$.
- For every type a, an infinite set of variables $v_{n,a}$ for each natural number n and a (possibly empty) set of constants c_a.
- The quantifiers $\forall$ and $\exists$, the equality symbol $=$, the connectives $\neg$, $\vee$, $\wedge$, $\rightarrow$.
- Predicates symbols and function symbols of any arity n.

Next, we introduce the set of types that are going to 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. We will follow the principles presented in [14], where D_a represents the set of possible objects (*denotations*) that describe the meanings of expressions of type a. Thus, denotations are the objects of the language that correspond to a given type.

The *set of types* Δ is defined recursively as follows:

- e is a type.
- t is a type.
- If a and b are types, then $(a \to b)$ is a type.

Let E be a given domain of entities. Then the semantics of the types are defined as:

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

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

We continue with the definitions for FOL typed term and FOL typed λ-calculus formula. A *FOL atomic term* is a constant or a variable of any type a. If $t_1, ..., t_n$ are FOL atomic terms, and f is a function symbol, $f(t_1, ..., t_n)$ is also a FOL atomic term of type e.

Definition 1. *The elements of the set Δ_α of FOL typed terms of type α are inductively defined as follows:*

1. *For each type a, every FOL atomic term of type a belongs to Δ_a.*
2. *If P is a predicate symbol of arity n and $t_1, ..., t_n$ are FOL atomic terms, then $P(t_1, ..., t_n)$ is a FOL typed atomic formula that belongs to Δ_t.*
3. *For any types a and b, if $\alpha \in \Delta_{a\to b}$ 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 $\alpha \in \Delta_b$ 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 λu.*[1]
5. *If $\alpha \in \Delta_t$ and $\beta \in \Delta_t$, then $\alpha \vee \beta$, $\alpha \wedge \beta$ and $\alpha \to \beta \in \Delta_t$.*
6. *If $\alpha \in \Delta_t$, then $\neg\ \alpha \in \Delta_t$.*
7. *If α 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 $\exists u\alpha$ $\in \Delta_t$.*

A *typed FOL λ-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 λ-calculus formula is in β-normal[2] form and there are no more λ-operators. They also correspond to the definition of closed and λI terms from classical lambda calculus theory.

Let us show an example. Consider the typed FOL formula $\lambda x.person(x)$, which has type $e \to t$. In this case, the variable x is a typed variable ranging

[1] An occurrence of a variable x in a typed term P is bound iff it is in the scope of an occurrence of λx in P, otherwise it is free. Occurrence is defined in Definition 2.

[2] A typed FOL λ-calculus formula is in β-normal form if no β-redex occurrence is possible. An example of a β-redex is a typed term of the form $(\lambda v.v)@(John)$.

over the entities of the domain and therefore it has type e. When an individual from the domain, such as the typed constant of type e, "John", is applied to the formula, we obtain a First Order Formula $person(John)$ which is of type t. The formula is no more than a function from individuals to truth values.

This illustrates the signature for Typed FOL λ-calculus and shows how we can specify a model with a domain of entities and a function to assign semantics to elements of the signature. With such a model, we can choose an interpretation for the formula $\lambda x.person(x)$ where only those entities of the domain that belong to the set of the predicate *person* would return the value *true* as output of the function. Therefore the interpretation of $\lambda x.person(x)@John$ would be the same as the one for $person(John)$ in the model. This is assured by the well-typed application that takes place between the formula $e \rightarrow t$ and the argument e, where a well-typed application is one where the type of the argument is the same as the "input" type of the function receiving it. In general, types are in charge of regulating which applications are possible and when both argument and function have the correct types, we have a *well-typed* expression.

We end this sub-section with three more definitions that will be used by the Inverse λ-Algorithms.

Definition 2 (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 ϕ or P occurs in ψ, then P occurs in $\phi \vee \psi$, $\phi \wedge \psi$, $\phi \rightarrow \psi$.
- If P occurs in ϕ, then P occurs in $\neg\phi$.
- If P occurs in ϕ, then P occurs in $\forall u \phi$ and $\exists u \phi$.
- If P occurs in ϕ or P occurs in ψ, then P occurs in $\phi = \psi$.
- If P occurs in any typed term t_i, then P occurs in $f(t_1, ..., t_n)$. Where f is a predicate symbol or a function symbol.

A *sub-term* of a λ-calculus term F is any term P that occurs in F [13].

Example 1. Consider the typed FOL λ-calculus formula $J = \lambda w.\lambda u.(man(John) \wedge w@\lambda z.(loves(z,u)))$. The typed terms $loves(z,u)$, $man(John)$, $\lambda z.(loves(z,u))$, and w occur in J, and hence, are some of the sub-terms of J. However, λw is not a sub-term.

Definition 3 (FOL λ-component). *Constants, quantifier variables, connectives $\neg$, $\wedge$, $\vee$ and $\rightarrow$, quantifiers $\exists$ and $\forall$, predicates and function symbols, and the equality symbol $=$ are denoted as FOL λ-components. The set of FOL λ-components of a formula J is identified as LC(J).*

Basically, all elements of the typed FOL λ-calculus signature except for the λ-application symbol, λ-abstractor and their corresponding bound variables, are considered FOL λ-components.

Example 2. Consider the typed FOL λ-calculus formula $J = \lambda w.\lambda u.(man(John) \wedge w@\lambda z.(loves(z,u)))$. $LC(J) = \{man, John, \wedge, loves\}$.

2.2 Type Order

So far we have introduced typed λ-calculus and the different types that will be assigned to typed terms. We now define the notion of an order that is associated with these types and that separates typed λ-calculus formulas to several classes. We will use the notion of orders to show that the Inverse λ-Algorithm is a complete algorithm for typed λ-calculus formulas up to order two. We start with a definition of type order and some intuition behind it:

Definition 4. *The order of a type is defined as:*

- *Base types e and t have order 0.*
- *For function types, $order(a \rightarrow b) = max(order(a) + 1, order(b))$.*

This definition is almost identical to the one introduced in [15], where base types have order one. We, however, follow the intuition introduced in [2]. Some examples of typed FOL λ-calculus formulas of different orders are the following:

- Order zero: $man(Vincent)$. It has type t.
- Order one: $\lambda v.\lambda u.(v \rightarrow u)$. It has type $e \rightarrow (e \rightarrow t)$.
- Order two: $\lambda v.\lambda u.(v@Mia \wedge u@Vincent)$. It has type $(e \rightarrow t) \rightarrow ((e \rightarrow t) \rightarrow t)$.
- Order three: $\lambda w.(w@(\lambda z.(happy(z))))$. It has type $((e \rightarrow t) \rightarrow t) \rightarrow t$.

With these simple examples, one can see the intuition behind the order of typed FOL λ-calculus formulas. Formulas of order zero correspond to expressions with base types. Formulas of order one correspond to expressions which start with a series of λ-abstractors followed by a FOL formula with variables bound to the initial λ-abstractors.

Formulas of order two extend the expressions allowed in order one by including applications. Formulas of order zero can be applied to variables inside the formula. Formulas of order three extend those present in order two by allowing λ-abstractors inside the expression after the initial λ-abstractors. In this case, formulas of order one can be applied to variables, this is why now, we can find λ-abstractors at the beginning and in the middle of the formulas. These claims can be easily proved by contradiction using the given definitions.

3 The Inverse λ-Algorithms

We start this section by presenting a lemma and some properties that are important for the Inverse λ-Algorithms and that are helpful to understand its completeness. After this introduction, the Inverse λ-operators of the Algorithm will be defined along with the explanation of its different parts. We start by presenting a lemma based on Lemma 1B1.1 from [13].

Lemma 1. *Given typed FOL λ-calculus formulas H, G and F, if $G@F$ β-reduces*[3] *to H, then $LC(G@F) = LC(H)$.*

Proof. Our definition of typed FOL λ-calculus formulas implies that all variables that appear in a formula are bound (except quantifier variables). Therefore, when one formula is applied to another and β-contractions are performed, none of the λ-components of those formulas will be modified by the application due to the definition of application in λ-calculus. In case of clashes with quantifier variables, α-conversion[4] would be used as usual. $\square$

With this lemma, it can be shown that given typed FOL λ-calculus formulas H, G and F such that $H = G@F$ or $H = F@G$, any FOL λ-component of H must be contained in either F or G (since we obtain H from these two formulas), and it can not be the case that a FOL λ-component in F or G will not appear in H (since there would be no proper second formula in the application that gives us H). This characteristic of the structure of the application is an essential part of the Inverse λ-Algorithms and the way in which it constructs the missing formula out of the other two. Also, by the way we defined typed FOL λ-calculus formulas, we eliminated the case where we set F to be $\lambda v.H$ which does not provide any semantic meaning to the expression that F represents but would be a valid classic λ-calculus formula for the application. Our aim 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 us to obtain the semantics of H, we are providing F with a semantic representation that contains the value we are looking for.

3.1 The Inverse λ-Operators

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

- Let G, H and J represent typed λ-calculus formulas, J^1,J^2,...,J^n represent typed terms; and v, w and v_1,...,v_n represent variables. Typed terms that are sub-terms of a typed term J^i are denoted as J^i_k.

If the formulas being processed within the algorithm do not satisfy any of the conditions, then the algorithm returns *null*.

[3] A typed term P β-reduces to Q if one can obtain Q after a finite sequence of β-contractions from P allowing for substitution of bound variables. The typed term $(\lambda v.v)@John$ β-reduces to $John$. We refer the reader to [12, 13] for additional lambda calculus definitions.

[4] α-conversion allows one to change the bound variables of a λ-expression if doing so does not affect the meaning of the expression. For example $\lambda x.x$ α-converts to $\lambda y.y$.

Definition 5 (operator :). *Consider two lists (of same length) of typed λ-calculus formulas $A_1, ..., A_n$ and $B_1, .., B_n$, and a typed FOL λ-calculus formula H. The result of the operation $H(A_1, ..., A_n : B_1, .., B_n)$ is defined as:*

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

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.

Next, we present the definition of the two inverse operators:

Definition 6 ($Inverse_L(H, G)$). *Given G and H:*

1. *G is $\lambda v.v$*
   - *$F = \lambda v.(v@H)$*
2. *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 $\lambda v_1, ..., v_s.J^i(J_1^i, ..., J_m^i : v_{k_1}, ..., v_{k_m})$*[5] *with $1 \leq s \leq m$ and $\forall p$, $1 \leq k_p \leq s$.*
   - *$F = \lambda w.H(J^1, ..., J^n : (w@J_{k_1}^1, ..., @J_{k_m}^1), ..., (w@J_{k_1}^n, ..., @J_{k_m}^n))$ where each J_{k_p} maps to a different v_{k_p} in G.*
4. *H is $\lambda v_1, ..., v_i.J$ and $f(\sigma_{i+1}, ..., \sigma_s)$ is a sub-term of J, G is $\lambda w.J(f(\sigma_{i+1}, ..., \sigma_s) : w@\sigma_{k_1}@...@\sigma_{k_s})$ with $\forall p$, $i + 1 \leq k_p \leq s$.*
   - *$F = \lambda w \lambda v_1, ..., v_i.(w@\lambda v_{i+1}, ..., v_s.(f(\sigma_{i+1}, ..., \sigma_s : v_{k_1}, ..., v_{k_s})))$*

Definition 7 ($Inverse_R(H, G)$). *Given G and H:*

1. *G is $\lambda v.v@J$*
   - *$F = Inverse_L(H, J)$*
2. *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 such that for all i, $J^i(J_1^i, \dots, J_m^i) = J^1(J_1^1, \dots, J_m^1 : J_1^i, \dots, J_m^i)$ and G is $\lambda w.H(J^1(J_1^1, ..., J_m^1), ..., J^n(J_1^n, ..., J_m^n) : (w@J_{k_1}^1, ..., @J_{k_m}^1),$ $..., (w@J_{k_1}^n, ..., @J_{k_m}^n))$ for some permutation $\{k_1, \dots, k_m\}$ of $\{1, \dots, m\}$.*
   - *$F = \lambda v_{k_1}, ..., v_{k_m}.J^1(J_1^1, ..., J_m^1 : v_1, ..., v_m)$.*
4. *If 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 $\lambda w.\lambda v_1,..., v_i.(w@\lambda v_{i+1},..., v_s.(J^1(J_{i+1}^1,..., J_s^1 \ : \ v_{k_1},..., v_{k_s})))$ with $\forall p$, $i + 1 \leq k_p \leq s$.*
   - *$F = \lambda w.J(J^1(J_{i+1}^1,..., J_s^1) : w@J_{k_1}^1@...@J_{k_s}^1)$*

[5] 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.

4 Examples

In this section we illustrate with examples each of the different conditions of the two algorithms. In each example λ-calculus formulas G and H are given and we want to find F using the given case of the Inverse λ-Algorithms. This section shows how the algorithms apply pattern matching to calculate F in the various cases. A use case example follows in the next section.

Inverse$_{\mathbf{L}}$ - Case 1:
$H = woman(Mia)$ and $G = \lambda x.x$. Then,
$F = \lambda x.(x@H)$, and in this case $F = \lambda x.(x@woman(Mia))$.
To demonstrate correctness we now apply G to F to get H:
$F \ @ \ G = \lambda x.(x@woman(Mia)) \ @ \ \lambda x.x$
$\quad = (\lambda x.x@woman(Mia)) = woman(Mia) = H.$

Inverse$_{\mathbf{L}}$ - Case 2:
$H = \lambda u.(man(Vincent) \wedge u \rightarrow man(Vincent))$ and $G = man(Vincent)$.
G is a sub-term of H. Therefore,
$F = \lambda v.H(G : v)$, which in this case yields $F = \lambda v.H(man(Vincent) : v)$.
We substitute every appearance of G in H by the variable "v" obtaining
$F = \lambda v.\ \lambda u.(v \wedge u \rightarrow v)$.

Inverse$_{\mathbf{L}}$ - Case 3:
$H = \lambda u.(woman(Mia) \wedge happy(Mia) \wedge man(Vincent) \wedge happy(Vincent)$
$\wedge\, loves(Mia, u))$, and $G = \lambda v.\lambda w.(v \wedge happy(w))$
G is not $\lambda v.v$. That satisfies the first condition.
To match the formula of G, the following are assigned the subterms of H:
$J^1 = woman(Mia) \wedge happy(Mia)$
$\quad J_1^1 = woman(Mia)$ and $J_2^1 = Mia$ (from $happy(Mia)$).
$J^2 = man(Vincent) \wedge happy(Vincent)$
$\quad J_1^2 = man(Vincent)$ and $J_2^2 = Vincent$ (from $happy(Vincent)$).
That satisfies second condition.
The third condition is satisfied since $\forall J^i \in H$, $G = \lambda v_1.\lambda v_2.J^i(J_1^i, J_2^i : v_1, v_2)$.
For example, for J^1:
$\quad G = \lambda v_1.\lambda v_2.J^1(woman(Mia), Mia : v_1, v_2) = \lambda v.\lambda w.(v \wedge happy(w))$.
Thus, $F = \lambda x.H((J^1 : x@J_1^1@J_2^1), (J^2 : x@J_1^2@J_2^2))$, which yields
$F = \lambda x.H((J^1 : x@woman(Mia)@Mia), (J^2 : x@man(Vincent)@Vincent))$,
$F = \lambda x.\lambda u.(x@woman(Mia)@Mia \wedge x@man(Vincent)@Vincent \wedge loves(Mia, u))$.

Inverse$_{\mathbf{L}}$ - Case 4:
$H = \lambda u.(happy(Mia) \rightarrow lovesBefore(Mia, Vincent, u))$
$G = \lambda v.(happy(Mia) \rightarrow v@Vincent@Mia)$.
$H = \lambda v_1.J$ with $J = happy(Mia) \rightarrow lovesBefore(Mia, Vincent, v_1)$ and
$\quad f(\sigma_2, ..., \sigma_s) = lovesBefore(Mia, Vincent, u)$ where
$\quad \sigma_2 = Mia$ and $\sigma_3 = Vincent$, chosen to match the variable v in G.
Therefore, G can be seen to be $\lambda v.J(f(\sigma_{1+1}, \sigma_3) : v@\sigma_{k_{1+1}}@\sigma_{k_3})$.

Thus, $G = \lambda v.J(f(\sigma_2, \sigma_3) : v@\sigma_3@\sigma_2)$, which in this case yields
$G = \lambda v.J(f(Mia, Vincent) : v@Vincent@Mia)$.
Thus, both conditions are satisfied. So, F can be calculated as follows:
$F = \lambda w \lambda v_1(w@\lambda v_{1+1}, ..., v_3.(f(\sigma_{1+1}, ..., \sigma_3 : v_{k_{1+1}}, ..., v_{k_3})))$,
$F = \lambda w \lambda v_1(w@\lambda v_2, v_3.(f(\sigma_2, \sigma_3 : v_{k_2}, v_{k_3})))$,
$F = \lambda w \lambda v_1(w@\lambda v_2, v_3.(f(\sigma_2, \sigma_3 : v_3, v_2)))$,
$F = \lambda w.\lambda v_1.(w@\lambda v_2, \lambda v_3.lovesBefore(v_3, v_2, v_1))$.

Inverse$_\mathbf{R}$ - Case 3:
$H = loves(Mia, Vincent) \wedge loves(Mia, Robert)$
$G = \lambda v.(v@Mia@Vincent \wedge v@Mia@Robert)$
G is not $\lambda v.v@J$, since $J = Mia@Vincent \wedge v@Mia@Robert$ cannot be a formula by the definition. That satisfies the first condition.
From H, and chosen to match the input to the variable v of G, one has the following formulas that are sub-terms:
$J^1 = loves(Mia, Vincent)$
$J^1_1 = Mia$, $J^1_2 = Vincent$
$J^2 = loves(Mia, Robert)$
$J^2_1 = Mia$, $J^2_2 = Robert$
Therefore, G can be seen to be of the form:
$G = \lambda x.H((J^1(J^1_1, J^1_2) : x@J^1_{k_1}@J^1_{k_2}), (J^2(J^2_1, J^2_2) : x@J^2_{k_1}@J^2_{k_2})$,
$G = \lambda x.H((J^1(J^1_1, J^1_2) : x@J^1_1@J^1_2), (J^2(J^2_1, J^2_2) : x@J^2_1@J^2_2)$,
$G = \lambda x.H((loves(Mia, Vincent) : x@Mia@Vincent), (loves(Mia, Robert) : x@Mia@Robert))$. Thus, the second condition of case 3 is satisfied.
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 $F = \lambda v_1.\lambda v_2.loves(v_1, v_2)$

5 Use Case Example

In this section, we revisit our earlier example and give another example to show how we can use the Inverse λ-Algorithms to obtain semantic representations for words when we have a sentence with its logical representation in typed first-order logic λ-calculus, the syntactic categories from CCG parsing and some semantic information. We again use an example from the database querying domain of [7].

Let us start with the sentences: "Texas borders a state.", and "What state borders Texas?". We will consider that our initial lexicon includes the semantics for simple nouns and verbs. If we obtain the output of a simplified CCG parsing with two categories "S" and "NP", and we add the semantics from our lexicon, we obtain what is presented in Table 2.

One can see in Table 2 that we are missing the semantic representation for the words "a" and "What". These two words are not part of our initial lexicon but using the Inverse λ-Algorithms we will be able to compute their corresponding typed first-order λ-calculus representation. Let us start with the first sentence.

Texas	borders	a	state
NP	$(S\backslash NP)/NP$	NP/NP	NP
NP	$(S\backslash NP)/NP$	NP	
NP	$(S\backslash NP)$		
S			

What	state	borders	Texas
$S/(S\backslash NP)/NP$	NP	$(S\backslash NP)/NP$	NP
$S/(S\backslash NP)/NP$	NP	$S\backslash NP$	
$S/(S\backslash NP)$		$S\backslash NP$	
S			

Texas	borders	a	state
texas	$\lambda w.\lambda x.(w@\lambda y.borders(y,x))$	???	$\lambda x.state(x)$
texas	$\lambda w.\lambda x.(w@\lambda y.borders(y,x))$	???	
texas	???		
$\exists x.[state(x) \wedge borders(x, texas)]$			

What	state	borders	Texas
???	$\lambda x.state(x)$	$\lambda y.\lambda x.borders(x,y)$	texas
???	$\lambda x.state(x)$	$\lambda x.borders(x, texas)$	
???		$\lambda x.borders(x, texas)$	
$\lambda x.state(x) \wedge borders(x, texas)$			

Table 2. CCG and λ-calculus derivations for "Texas borders a state.","What state borders Texas?"

We can take the meaning of the sentence and the meaning of "Texas" to calculate the representation of "borders a state".

We can see in the CCG tree that "borders a state" has category $(S\backslash NP)$ and therefore the λ-calculus expression will receive the word "Texas" from the left. Following this, if we take H as the meaning of the sentence and G as the meaning of "Texas", we can use $Inverse_L(H, G)$ to obtain the expression for "borders a state". In this case, option one of the algorithm is satisfied with $texas$ as formula J and $F = \lambda y.\exists x.[state(x) \wedge borders(x, y)]$.

Now we have the expression for "borders a state" and "borders", we 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 we obtain F as $\lambda y.\exists x.[state(x) \wedge y@x]$. Finally, we call $Inverse_L(H, G)$ with H being the expression for "a state" and G being "state" to obtain the desired representation for "a". In this case, option two of the algorithm is satisfied and $F = \lambda z.\lambda y.\exists x.[z@x \wedge y@x]$. 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 shown above. First we call $Inverse_L(H, G)$ with H being the meaning of the sentence and G being "borders Texas" to obtain the meaning of "What state". Option two of the algorithm is satisfied and $F = \lambda y.\lambda x.(state(x) \wedge y@x)$. Next, we call $Inverse_L(H, G)$ again with "What state" and "state" to obtain the desired meaning of "What". Option two of the algorithm is satisfied and $F = \lambda z.\lambda y.\lambda x.(z@x \wedge y@x)$.

Using the Inverse λ-Algorithms we have easily added to our lexicon the λ-calculus representation for the words "a" and "What".

6 Theorems

Theorem 1 (Soundness). *Given two typed FOL λ-calculus formulas H and G in β-normal form, if $Inverse_L(H, G)$ returns a non-null value F, then $H = F \;@\; G$.*

Theorem 2 (Soundness). *Given two typed FOL λ-calculus formulas H and G in β-normal form, if $Inverse_R(H, G)$ returns a non-null value F, then $H = G \;@\; F$.*

Theorem 3 (Completeness). *For any two typed FOL λ-calculus formulas H and G in β-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 FOL λ-calculus formulas Θ_F of order two or less in β-normal form, such that $\forall F_i \in \Theta_F$, $H = F_i @G$, then $Inverse_L(H, G)$ will give us an F where $F \in \Theta_F$.*

Theorem 4 (Completeness). *For any two typed FOL λ-calculus formulas H and G of order two or less in β-normal form, if there exists a set of typed FOL λ-calculus formulas Θ_F of order one or less in β-normal form, such that $\forall F_i \in \Theta_F$, $H = G@F_i$, then $Inverse_R(H, G)$ will give us an F, where $F \in \Theta_F$.*

Theorem 5 ($Inverse_L$ complexity). *The $Inverse_L$ Algorithm runs in exponential time in the number of variables in G and polynomial time in the size of the formulas H and G.*

Theorem 6 ($Inverse_R$ complexity). *The $Inverse_R$ Algorithm runs in exponential time in the number of variables in G and polynomial time in the size of the formulas H and G.*

Due to space constraints, we will only comment on how the complexity, soundness and completeness proofs are structured. $Inverse_L$ has worst-case complexity in case 3. For every subterm of H it is necessary to check if a partial permutation of its subterms can be used to generate G. If G has k variables, then the number of permutations that needs to be checked is in worst case $\frac{n!}{(n-k)!}$ where n is the length of H. When k is small, as it is in most of the applications we have tested (see Section 8 below), the number of permutations becomes approximately $O(n^k)$.

The complexity of $Inverse_R$ is the same as $Inverse_L$ since $Inverse_L$ is called as a subroutine in case 1. All other cases of $Inverse_R$ run in polynomial time. Note that case 3 of $Inverse_R$ has polynomial complexity since the formulas given as input to the applications in G can be used to find the subterms of H that can generate them. Therefore, we do not have to search all permutation of the subterms as in $Inverse_L$.

The soundness proof shows how in each of the four cases of $Inverse_L$, the typed FOL λ-calculus formula H is obtained by applying F to G. The application $F@G$ is computed using the expressions from the algorithm for F and G, generating the expression for H given in the algorithm. The same reasoning is followed for $Inverse_R$.

The completeness proof is divided to six cases, which correspond to the six possible valid combinations of orders that H, F and G may have, such that the order of the terms will be less than 2. These are shown in Table 3. For each case, it is proven by contradiction that $Inverse_L$ and $Inverse_R$ return a formula F if one such F exists. It is done by assuming that they return a *null* value and reaching a contradiction at the end of the proof. In the process, each of the four conditions of the algorithms are analyzed, where it is shown that at least one of the conditions of the algorithm has been satisfied for each of the six cases.

H	F	G	FOL type examples for formula F
0	1	0	$e \to t$
1	1	0	$e \to (e \to t)$
2	2	0	$e \to ((e \to t) \to t)$
0	2	1	$(e \to t) \to t$
1	2	1	$(e \to t) \to (e \to t)$
2	2	1	$(e \to t) \to ((e \to t) \to t)$

Table 3. Possible order combinations for F, G and H formulas, with $H = F@G$.

7 Related Work

The problems solved by the Inverse λ-Algorithms are similar to two problems referred to in the literature as "higher-order matching" and "Interpolation problem". This problem consists of determining if a λ-calculus term, in the simply typed λ-calculus, is an instance of another. It can also be understood as solving the equation $M = N$ where M and N are simply typed λ-terms and N is a closed[6] term. More intuitively, the problem is to find a substitution σ 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 [16]. 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 [16] has to be understood as second order in this research.

The higher-order matching and interpolation problems are defined with respect to an extension of λ-calculus denoted as $\beta\eta$-calculus[7], while the Inverse λ-Algorithms follow the theory of β-calculus, since it is the theory most commonly used in linguistics. Another important difference is that in the case of $Inverse_L$ and $Inverse_R$, terms G,F and H are typed λ-calculus formulas that, by definition, have been set as closed and λI terms, and are also considered in β-normal form. The higher-order matching and interpolation problems do

[6] A λ-expression is closed if no free variables occur in it

[7] $\beta\eta$-calculus allows for η-conversions as well as β-reductions. An η-conversion converts between $\lambda x.f@x$ and f whenever x does not appear as a free variable in f.

not enforce these restrictions in the terms involved. However, in essence, both approaches deal with the same problem. Thus, we argue that $Inverse_R$ and $Inverse_L$ can be considered special cases of the matching and interpolation problems , respectively (when we restrict the problem to β-calculus and only try to substitute for one variable in M).

However, the Inverse λ-Algorithms provide an important contribution. As it is stated in [15], the higher-order matching problem, in the general case, is undecidable when using β-calculus, as shown in [17]. $Inverse_R$ is an algorithm for computing a subset of the third order matching problem under β-calculus. Also, the higher-order matching algorithm proposed in [16], 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.

Another related work, [18, 19], discusses two algorithms that solve the problem of learning word-to-meaning mappings extended to the field of typed λ-calculus. In [19], the algorithm introduced considers β-calculus and closed λI terms. This definition is very closely related to the $Inverse_R$ definition. However, the author refers to the problem that the algorithm presents with incompatibility between meaning of words already known and meaning of words obtained by the algorithm. Our approach does not present such problem. Another key difference refers to the assumptions made before executing the algorithm. In [18, 19], there is a first phase where the learner, in some way, obtains the set of constants that form each word. In the case of the Inverse λ-Algorithms, 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.

The comparison with the works [15, 16, 18] is leading us to research the possible benefits of defining the Inverse λ-Algorithms in terms of $\beta\eta$-calculus, in order to use useful related results. This step will be approached in future work.

A more general notion of higher-order matching is the notion of “higher-order unification”. The difficulty of higher-order unification is studied in [20, 21]. In [22] a restricted version of higher-order unification is defined and used. They describe an algorithm which given h, finds pairs of logical expressions (f, g) such that either $f(g) = h$ or $\lambda x.f(g(x)) = h$. Note that in our Inverse λ-Algorithms the input consists of two expressions and the output is a single expression.

8 Implementation Success and Evaluation

The usefulness of the two Inverse λ-Algorithms is validated in the natural language learning system [9], which learns the semantic representations of words from sentences. The algorithms were implemented by those authors as part of that system. The results obtained by the system in [9] are quite promising. A summary is presented in Table 4. The system outperforms earlier systems with respect to the F-measure for the two standard corpora GEOQUERY and CLANG.

GEOQUERY	Precision	Recall	F-measure
Inverse+	93.41	89.04	91.17
Inverse	91.12	85.78	88.37

CLANG	Precision	Recall	F-measure
Inverse+i	87.67	79.08	83.15
Inverse+	85.74	76.63	80.92

Table 4. Performance results for GEOQUERY and CLANG.

In the table, $Inverse$ and $Inverse+$ are two different versions of the system. As explained in [9], $Inverse$ uses the two Inverse λ-Algorithms, $Inverse_L$ and $Inverse_R$ and a process called *generalization*, which generalizes the meaning of unknown words from known words based on the syntactic information received from the CCG parser. $Inverse+$ adds trivial inverse results for some words and "on demand" *generalization*. $Inverse+i$ considers the semantic representations "definec" and "definer" of the training data as the same element, with respect to the CLANG corpus.

The evaluation of the data was performed using 10 fold cross validation and the $C\&C$ parser from [6]. An equal number of train and test sentences were arbitrarily selected from the GEOQUERY and CLANG corpora. For more details about this system and its results, we refer the reader to [9].

9 Conclusion

In this work, we have presented two Inverse λ-Algorithms and have shown their application to typed first-order logic λ-calculus. With this algorithm we are able to automatically obtain semantic representations of unknown words using the information already available from known sentences and words. These two algorithms not only work for first-order logic, but with minor changes work for a group of other Knowledge Representation languages such as Answer Set Programming, and Linear Temporal Logic and Dynamic Logic; but formal results with respect to those languages need to be proven. For the algorithms we have developed a completeness proof up to second order expressions. Blackburn and Bos in [2] mention that natural language semantics rarely requires types above order three. Completeness results for third order expressions is the next step to extend this work and is part of our immediate future plans.

Acknowledgement

We thank Vladimir Lifschitz, Yuliya Lierler and Johan Bos for introducing us to the use of λ-calculus in characterizing natural language. We thank Juraj Dzifcak for implementing the inverse λ-Algorithms, building the system mentioned in Section 8 and doing the evaluations mentioned in that section. We acknowledge support by NSF (grant number 0950440), IARPA and ONR-MURI for this work.

References

1. Gamut, L.: Logic, Language, and Meaning. The University of Chicago Press (1991)

2. Blackburn, P., Bos, J.: Representation and Inference for Natural Language: A First Course in Computational Semantics. Center for the Study of Language (2005)
3. Zettlemoyer, L.S., Collins, M.: Online learning of relaxed ccg grammars for parsing to logical form. In: Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning. (2007) 678–687
4. Baral, C., Dzifcak, J., Son, T.C.: Using answer set programming and lambda calculus to characterize natural language sentences with normatives and exceptions. In: AAAI'08: Proceedings of the 23rd national conference on Artificial intelligence. (2008) 818–823
5. Dzifcak, J., Scheutz, M., Baral, C., Schermerhorn, P.: What to do and how to do it: translating natural language directives into temporal and dynamic logic representation for goal management and action execution. In: Robotics and Automation, 2009. ICRA '09. (2009) 4163 –4168
6. Clark, S., Curran, J.R.: Wide-coverage efficient statistical parsing with ccg and log-linear models. Computational Linguistics **33** (2007)
7. Zettlemoyer, L.S., Collins, M.: Learning to map sentences to logical form: Structured classification with probabilistic categorial grammars. In: 21th Annual Conference on Uncertainty in Artificial Intelligence. (2005) 658–666
8. Steedman, M.: The syntactic process. MIT Press (2000)
9. Baral, C., Dzifcak, J., Gonzalez, M.A., Zhou, J.: Using inverse lambda and generalization to translate english to formal languages. In: Proceedings of the 9th International Conference on Computational Semantics (ICWS 2011), to appear. (2011)
10. Montague, R.: Formal Philosophy. Selected Papers of Richard Montague. New Haven: Yale University Press (1974)
11. Barendregt, H.: Lambda Calculi with Types, Handbook of Logic in Computer Science. Volume II. Oxford University Press (1992)
12. Hindley, J.: Introduction to Combinators and Lambda-Calculus. Cambridge University press (1986)
13. Hindley, J.: Basic Simple Type Theory. Cambridge University press (1997)
14. Barbara H. Partee, A.T.M., Wall, R.E.: Mathematical Methods in Linguistics. Kluwer Academic Publishers (1990)
15. Stirling, C.: Decidability of higher-order matching. To appear Logical Methods in Computer Science **5**(3) (2009)
16. Dowek, G.: Third order matching is decidable. Annals of Pure and Applied Logic **69**(2-3) (1994) 135–155
17. Loader, R.: Higher-order beta-matching is undecidable. Logic Journal of IGPL **11**(1) (2003) 51–68
18. Kanazawa, M.: Learning word-to-meaning mappings in logical semantics. In: Proceedings of the Thirteenth Amsterdam Colloquium. (2001) 126–131
19. Kanazawa, M.: Computing word meanings by interpolation. In: Proceedings of the Fourteenth Amsterdam Colloquium. (2003) 157–162
20. Huet, G.: The undecidability of unication in third order logic. Information and Control **22**(3) (1973) 257–267
21. Huet, G.: A unication algorithm for typed calculus. Theoretical Computer Science **1** (1975) 27–57
22. Kwiatkowski, T., Zettlemoyer, L., Goldwater, S., Steedman, M.: Inducing probabilistic ccg grammars from logical form with higher-order unification. In: Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP). (2010)