Robust implementation of NL2KR System and it's application in iRODS domain

by

Kanchan Ravishankar Kumbhare

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

Approved June 2013 by the
Graduate Supervisory Committee:

Chitta Baral, Chair
Baoxin Li
Jieping Ye

ARIZONA STATE UNIVERSITY

August 2013

ABSTRACT

Currently, to interact with computer based systems one needs to learn the specific interface language of that system. In most cases, interaction would be much easier if it could be done in natural language. For that, we will need a module which understands natural language and automatically translates it to the interface language of the system. NL2KR (Natural language to knowledge representation) v.1 system is a prototype of such a system. It is a learning based system that learns new meanings of words in terms of λ-calculus formulas given an initial lexicon of some words and their meanings and a training corpus of sentences with their translations. As a part of this thesis, we take the prototype NL2KR v.1 system and enhance various components of it to make it usable for somewhat substantial and useful interface languages. We revamped the lexicon learning components, Inverse-λ and Generalization modules, and redesigned the lexicon learning algorithm which uses these components to learn new meanings of words. Similarly, we re-developed an inbuilt parser of the system in Answer Set Programming (ASP) and also integrated external parser with the system. Apart from this, we added some new rich features like various system configurations and memory cache in the learning component of the NL2KR system. These enhancements helped in learning more meanings of the words, boosted performance of the system by reducing the computation time by a factor of 8 and improved the usability of the system.

We evaluated the NL2KR system on iRODS domain. iRODS is a rule-oriented data system, which helps in managing large set of computer files using policies. This system provides a Rule-Oriented interface langauge whose syntactic structure is like any procedural programming language (eg. C). However, direct translation of natural language (NL) to this interface language is difficult. So, for automatic translation of NL to this language, we define a simple Intermediate Policy Declarative Language (IPDL) to represent the knowledge in the policies, which then can be directly translated to iRODS rules. We develop a corpus of 100 policy statements and manually translate them to IPDL langauge. This corpus is then used for the evaluation of NL2KR system. We performed 10 fold cross validation on the system. Furthermore, using this corpus we illustrate how different components of our NL2KR system work.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my supervisor Dr. Chitta Baral for his valuable comments, remarks and guidances throughout my thesis. It is through him that I started learning about research. I would also like to thank Nguyen Vo for helping me in most of the work on my thesis and always providing his valuable comments. I would also like to thank all my colleagues in BioAI lab for their support.

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

Table Page

LIST OF FIGURES

Chapter 1

Introduction and Motivation

For the purpose of interaction with computer based systems, we need to learn the specific interface language of these systems. If a user wants to use one such system he should have good knowledge of the grammar of the system's interface language. In most cases, this interaction would be much easier if it would be done in natural language.

In order to automate the process of direct interaction with the computer based systems in natural language, we need a module which can translate natural language to an interface language of these systems. As there is no single interface language which is suitable for different systems, we need a module which should be able to learn to translate natural language to various interface languages. In [1], a prototype system is developed for this task. This prototype uses an initial lexicon containing some words and their meanings and a set of training corpus containing sentences in natural language and their translations to learn new meanings of words. The system then uses the new learned lexicon to translate new sentences. In this research, we make significant enhancements in this prototype by augumenting various components and adding various new powerful features. We make various improvements in learning component of the system, specifically in natural language text parser, lexicon learning components i.e. Inverse-λ and Generalization, learning algorithm among others. Some of the new features added to the system are memory cache to improve the computation time, better system configuration to provide flexibility in efficiently handling the system.

We evaluate the NL2KR system on the iRODS[1] system which is used to manage large set of computer files using policies. This system provides Rule-Oriented language [13] to interact with users and accordingly responds to their requests. The syntactic structure of this language is very similar to a procedural programming language like C. A direct translation of natural language to this interface language would be difficult. We design a simple Intermedicate Policy Declarative language (IPDL) such that natural language can be translated to this

[1] https://www.irods.org

language and further one can directly translate IPDL to iRODS rules. We develop a corpus size of 100 policies and manually translate them to IPDL statements for the evaluation purpose. A 10-fold cross validation is performed on the system using this corpus. We also illustrate working of various components of the NL2KR system using this corpus.

1.1 Existing Work and Related Research

NL2KR v.1 System

The main idea behind the implementation of the prototype NL2KR v.1 system is to be able to translate the natural language text into an appropriate knowledge representation language so that a reasoning engine can reason with it and give a response accordingly. This system is based on the approach described by Montague [10] whose idea was to represent meanings of words/phrases in natural language text in terms of λ-calculus expressions such that by compositions of the meanings of the constituent words, we can obtain the meaning of natural language text. But, in order to compose the meaning of a sentence, we will need meanings of each word/phrase in a sentence. Manually defining such meanings would be a very tedious task because some meanings might have complex λ-calculus formulas. This system uses two very elegant approaches: Inverse-λ and generalization for the automation of obtaining the λ-calculus meanings of various words and phrases. In this section, we will briefly describe the NL2KR v.1 system.

Overview of the system

The NL2KR system has two sub-parts which depends on each other (1) NL2KR-L, builds a final lexicon by learning λ-calculus formulas of words in training dataset sentences. A weight is assigned to each λ-calculus formula to deal with multiple meanings of a word. (2) NL2KR-T, which uses the learned lexicon from NL2KR-L and translates a given sentence to a formal representation. However in v.1 of the system, these components were integrated together into a single application system.

The system takes an initial lexicon consisting of some words and their meanings in terms of λ-calculus expressions, a set of training sentences and their formal representations, and a set of test sentences and their formal representations as an input. The learning sub-part of the system uses Inverse-λ and Generalization algorithms to learn meanings of new encountered words which are not present in initial lexicon and adds them to the lexicon. A parameter learning method is then used to estimate a weight for each lexicon entry (word, its syntactic category and meaning) such that the probability that each sentence is translated to given formal representation is maximized. Basic translation methodology is based on the technique discussed in [1] i.e. Probabilistic Combinatorial Categorical Grammars (PCCG). Given a sentence S, its translation M can be obtained by finding M which maximizes the probability function $P(M|S;\Theta)$, which denotes the probability of translation of S to M , given the parameter vector i.e. $argmax_M\ P(M|S;\Theta)$. As a particular translation can be obtained using multiple parse trees, $P(M|S;\Theta)$ is defined as $\sum_T\ P(M,T|S;\Theta)$. The probability $P(M,T|S;\Theta)$ is defined using the feature vector $\bar{f}(M,T,S)$ and a log linear model as follows:

$$P(M,T|S;\bar{\Theta}) = \frac{e^{\bar{f}(M,T,S).\bar{\Theta}}}{\sum_{(M,T)} e^{\bar{f}(M,T,S).\bar{\Theta}}}$$

The feature vector is the one that counts the number of times each lexical entry is used in a parse tree T. A lexicon Λ and a set of training data (S_i, M_i), $i = 1, .., n$, where S_i is a sentence and M_i is its translation are used to find Θ that maximizes $L(\Theta, \Lambda)$, given by

$$P(M_1|S_1;\Theta) \times P(M_2|S_2;\Theta) \times ... \times P(M_n|S_n;\Theta)$$

Using this parameter estimation method, the lexicon is updated by adding weights for each lexical entry. After parameter estimation, lexicon is again generalized to learn more meanings.

Once, the training component finishes its job, the translation component (NL2KR-T) uses this updated lexicon, a test data set and translates sentences in test data set using PCCG parser. Since words can have multiple meanings and their associated λ-calculus expressions, weights assigned to each lexical entry in the lexicon helps in deciding correct meaning of a word

in the context of a sentence such that the meanings which corresponds to maximum probability of translation are picked.

An overview of the prototype NL2KR v.1 system is described in figure 1.1. We will now elaborate on various modules in NL2KR system.

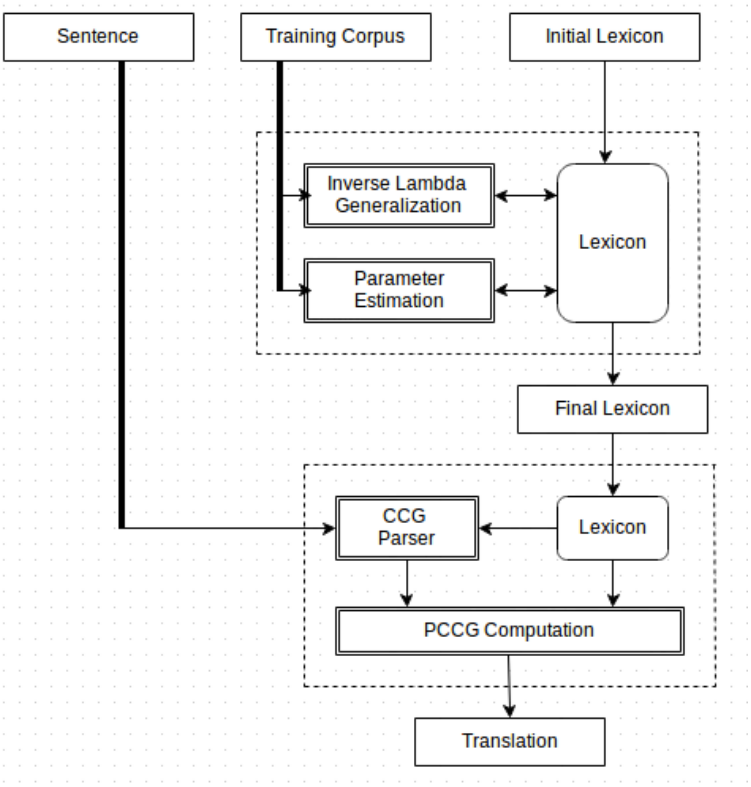


Figure 1.1: Overview of the prototype NL2KR v.1 system

Inverse-λ Operators

As mentioned earlier, Inverse λ-module learns new meanings of words in training sentences. For implementation of this module, theory proposed in [2] is used. The main idea behind this theory is to compute a λ-expression F such that $H = F@G$ or $H = G@F$. A basic version of two components $Inverse_L$ and $Inverse_R$ is implemented in NL2KR v.1.

We will now present the definition of these components from [2]. Before, defining these terminologies we will introduce some definitions and explanations necessary to help understand them. Different symbols used in the definition are as follows:

- $F,\ G,\ H$ and J represent the typed λ-calculus formulas.

- $J^1, J^2, .., J^n$ represent the typed terms.

- $v,\ w$ and $v^1, v^2, .., v^n$ represent the variables.

- Typed terms which are sub-terms of J^i are represented by J^i_k.

Definition 1 (Operator :) *If we have two lists (same length) of typed λ-calculus formulas $A_1, ..., A_n$ and $B_1, .., B_n$, and a typed λ-calculus formula H, then 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 step 2. Otherwise,stop.

Definition 2 ($Inverse_L(H, G)$) *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 $\lambda v.v$, $(J^1(J^1_1, ..., J^1_m), J^2(J^2_1, ..., J^2_m),\ ...\ , J^n(J^n_1, ..., J^n_m))$ 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 = $\lambda w.H(J^1, ..., J^n : (w@J^1_{k_1}, ..., @J^1_{k_m}), ..., (w@J^n_{k_1}, ..., @J^n_{k_m}))$ where each J_{k_p} maps to a different v_{k_p} in G.*
4. *H is $\lambda v_1, ..., v_i.J$ and $f(\sigma_{i+1}, ..., \sigma_s)$ 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 3 ($Inverse_R(H, G)$**)** *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* $\lambda v.v@J$*,* $(J^1(J^1_1, ..., J^1_m), J^2(J^2_1, ..., J^2_m),\ \ ... \ , J^n(J^n_1, ..., J^n_m))$ *are sub-terms of* H *such that for all* i*,* $J^i(J^i_1, \ldots, J^i_m) = J^1(J^1_1, \ldots, J^1_m : J^i_1, \ldots, J^i_m)$ *and* G *is* $\lambda w.H(J^1(J^1_1, ..., J^1_m), ..., J^n(J^n_1, ..., J^n_m) : (w@J^1_{k_1}, ..., @J^1_{k_m}),$ $..., (w@J^n_{k_1}, ..., @J^n_{k_m}))$ *for some permutation* $\{k_1, \ldots, k_m\}$ *of* $\{1, \ldots, m\}$*.*
   - F = $\lambda v_{k_1}, ..., v_{k_m}.J^1(J^1_1, ..., J^1_m : v_1, ..., v_m)$*.*
4. H *is* $\lambda v_1, ..., v_i.J$ *and* $f(\sigma_{i+1}, ..., \sigma_s)$ *is a sub-term of* J*,* G *is* $\lambda w.\lambda v_1, ..., v_i.(w@$ $\lambda v_{i+1}, ..., v_s.(f(\sigma_{i+1}, ..., \sigma_s : v_{k_1}, ..., v_{k_s})))$ *with* $\forall p$*,* $i + 1 \leq k_p \leq s$*.*
   - F = $\lambda w.J(f(\sigma_{i+1}, ..., \sigma_s) : w@\sigma_{k_1}@...@\sigma_{k_s})$

Generalization Algorithm

Inverse-L and Inverse-R are not enough to learn new meanings of words. Without any form of generalization the system will not be able to extend meanings of words to go beyond the ones that are present in the training data set. For generalization module in NL2KR v.1 system, the generalization technique described in [1] is implemented. The generalization algorithm is described as follows:

The $Generalize_D$ algorithm takes the lexicon Λ and a word entry α which we want to generalize. $IDENTIFY(l_i(word), l_i(semantics))$ method will identify the part of semantics of a lexical entry in the lexion such that the word in this lexical entry is present in the semantics. For. eg. $eats$ is part of semantics $\lambda y.\lambda x.eats(x, y)$. $REPLACE(l_i(semantics), I, \alpha(word))$ will replace the part in semantics of lexical entry with a word of α. For eg. We want to generalize

Algorithm 1 $Generalize_D(\Lambda, \alpha)$

for $l_i \in \Lambda$ **do**
  if $l_i(category) = \alpha(category)$ **then**
    $I = IDENTIFY(l_i(word), l_i(semantics))$
    $S = REPLACE(l_i(semantics), I, \alpha(word))$
    $\Lambda = \Lambda \cup (\alpha(word), \alpha(category), S)$
  end if
end for

the meaning of word entry ($plays$, $(S\backslash NP)/NP$). The lexicon already contains a lexical entry ($eats$, $(S\backslash NP)/NP$, $\lambda y.\lambda x.eats(x, y)$) whose category is same as that of word $plays$. The above algorithm will add a new lexical entry ($plays$, $(S\backslash NP)/NP$, $\lambda y.\lambda x.plays(x, y)$) to the lexicon by generalizing $plays$ from $eats$. Also, a default weight of 0.01 is assigned to the new lexical entry for $plays$.

Learning Algorithm

NL2KR-L component of existing system uses following learning algorithm.

Algorithm 2 Lexicon Learning

Input: Input: A set of training sentences with their corresponding desired formal representations $S = (S_i, L_i), i = 1, ..n$ where S_i is the training sentence and L_i is its formal representation, an initial feature vector Θ_0, an initial lexicon Λ_0. Initially default weight (0.01) is assigned to each lexical entry.

Output: Output: An updated Λ_{T+1} and an updated feature vector Θ_{T+1}

Set Λ_0
for $t = 1, ..T$ **do**
  for $i = 1, ..n$ **do**
    for $j = 1, ..n$ **do**
      $Traverse_j$: apply $Inverse_L$, $Inverse_R$ and $Generalize_D$ to find new λ-expressions of words or phrases α
      Set Λ_{t+1} $\Lambda_t \cup \alpha$
    end for
  end for
  Parameter Estimation
  Set $\Theta_{t+1} = UPDATE(\Theta_t, \Lambda_{t+1})$
end for
return $Generalize(\Lambda_T, \Lambda_T)$, Θ_T

Background Related Research

In this research, our main focus is on efficient implementation of various components of NL2KR system and evaluation of this system on iRODS domain. The paper [1] describes the basic implementation of components like Inverse-λ, generalization, Learning algorithm. In our work, we have made improvements in these components for effective training of the system. Also, we have implemented components: Inverse-λ , CCG parser and defined the representation of formalisms like λ-calculus expressions in Answer Set Programming (ASP) [3] because of its success in efficiently solving search problems which are reduced to computing stable models and it can efficiently represent knowledge that needs non-monotonic reasoning which can be used to improve performance of the system. Among many ASP systems, such as Smodels [7], DLV [8] and Clingo [9], we use the Clingo system because Lua [2], a scripting language which is required in implementation of some basic λ-calculus operations, is been demonstrably integrated with Clingo [9].

For the evaluation of this system on iRODS domain, we have defined a simple yet expressive Intermediate Policy Declarative Language (IPDL). The paper [5] provides description about Policy Description language which is based on event, condition and action paradigms. In [6], an Action Description Language is introduced consisting of value and effect propositions. The design of IPDL is strongly inspired by these researches.

1.2 Motivation

iRODS Corpus

In this work, we use the iRODS corpus which consists of policies/rules given in English language and their translations in Intermediate Policy Declarative language (IPDL). We have a corpus of 100 such policies in English language and manually translated these policies to IPDL statements. iRODS system provides an iRODS Rule Language [13] to define policies and

[2] http://www.lua.org/

actions in the system. An example of an iRODS policy in English is ‘Check if every file readable by Tom is also readable by Jim’. This policy can be written in iRODS Rule language as follows:

```
rule {
    *DataObjRS=readableDataObjs("Tom");
    *Suc=true;
    foreach(*DataObjKVP in *DataObjRS) {
        msiGetValByKey(*DataObjKVP, "DATA_NAME", *DataObjName);
        msiGetValByKey(*DataObjKVP, "COLL_NAME", *CollName);
        if(!canAccess(/*CollName/*DataName, "Jim", "read") {
            *Suc=false;
            *Break;
        }
     }
     *Suc;
}
```

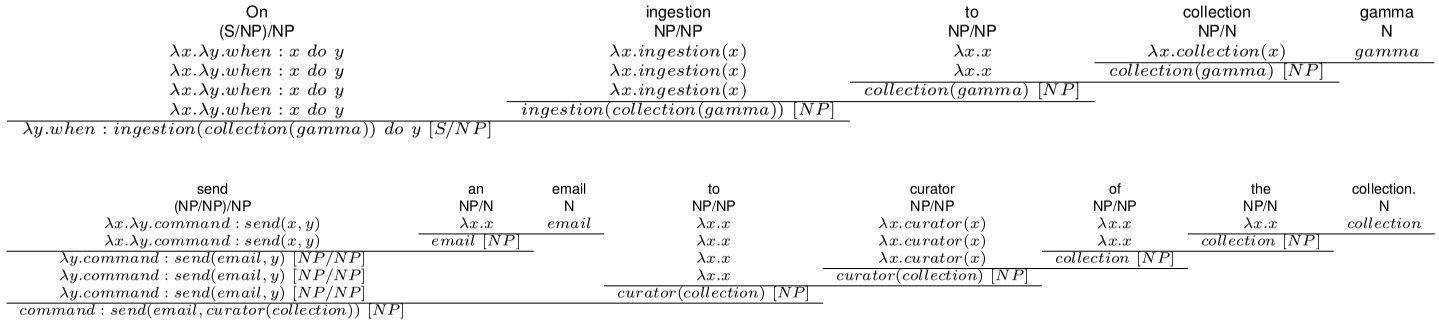
Motivation Example

In the previous section, we talked about iRODS corpus and an example of a policy in iRODS Rule Language. In order to automatically translate such policies, we need a system which should be able to (1). process the natural language, (2). extract the knowledge and (3). use the extracted knowledge to generate rules. However, it is difficult to have a system which directly translates policies in natural language (NL) to such rules. For translation of NL to iRODS rules, we will need an Intermediate Declarative Language such that we first translate natural language sentences to it and then further it can be translated to iRODS rules.

In this research, we designed a simple Intermediate Policy Declarative Language (IPDL) for iRODS system. An example of translation of policy ’On ingestion into collection gamma send an email to curator of the collection’ to IPDL using the semantics of words in the sentence in

terms of λ-calculus expressions is described in table 1.1. A detailed description of syntax of IPDL is given in chapter 2.

Table 1.1: CCG and λ-calculus derivation for ‘On ingestion to collection gamma send an email to curator of the collection’



1.3 Main Contribution

In this research, we have implemented NL2KR v.2 system over existing prototype NL2KR v.1 by improving various modules of the existing system. We have implemented an inbuilt CCG parser for the system in ASP and also integrated an existing CCG parser (ASPccgTk parser [14]) which also uses ASP to solve the parsing problem. We have also revamped the lexicon learning techniques like Inverse-λ and Generalization and learning algorithm of the system. Apart from this, we have also added some new features like better system configuration to efficiently handle the system, memory cache in the learning component which improved the computation time by approximately 8 times.

The existing system uses Java implementation of various components like Inverse-λ algorithm, and various λ-calculus operations. Even though Java is very good programming language, the type of problems we needed to solve as mentioned in the previous sections can be easily solved using ASP which is very efficient in solving a search problem with only few ASP rules. In order to implement Inverse-λ algorithm and various λ-calculus operations in ASP, we need a good representation of the λ-expressions in ASP whether it could be a FOL-λ-expression or an ASP-λ-expression or an IPDL-λ-expression. One of the main contributions of this research is introduction of a succinct representation of λ-calculus expressions in ASP. We have also implemented various λ-calculus operations and Inverse-λ in ASP. However, in order to have an efficient implementation of these techniques, we needed some extra functionality like generation of new IDs, string operations which ASP does not provide. So we intend to use a light-weight scripting language Lua which is already integrated with the existing ASP solver Clingo [9]. Thus, by using the power of ASP and Lua we are able to efficiently implement components mentioned above. The details of this implementation is explained in chapter 4.

Another major contribution of this research is introduction of an Intermediate Policy Declarative language (IPDL) for iRODS system. iRODS system is a rule oriented data management system used to manage large set of computer files. As we have already mentioned in previous section that this system provides a rule oriented language to interpret policies. We

have designed a simple policy declarative language which can be further translated to rules in iRODS rule oriented language. We need IPDL because we want to translate policies in natural language to iRODS rules which is impossible if we do a direct translation being a procedural language structure of iRODS rule oriented language.

We evaluated NL2KR v.2 system on iRODS domain. We first train the system using a set of policies in natural language (English) and their corresponding IPDL translations and then we test the system by translating new policies in natural language. A detailed description of evaluation of the system and it's performance analysis is provided in chapter 5.

1.4 Thesis outline

We have already talked about existing NL2KR v.1 system in this chapter. Rest of the work is organized as follows: Chapter 2 presents an introduction of iRODS system and syntax of IPDL language. Chapter 3 describes shortcomings of the NL2KR v.1 system and improvements done in various components and integration of some new components to NL2KR v.2 system. Chapter 4 talks about an implementation of components operating on λ-calculus expression in ASP+Lua. Chapter 5 describes evaluation of NL2KR v.2 system and analysis of performance of the system. Finally, chapter 6 finishes this work with conclusion and potential future improvements in the system.

Chapter 2

Policy Translation

2.1 Introduction to iRODS system

iRODS system is Integrated Rule Oriented Data system developed to provide flexibility, adaptability, customization of data. The iRODS system has a data grid architecture which is based on client/server model and distributed storage. This system has a database to maintain attributes and states of data and operations. It also provides a rule system which can be used to enforce and execute adaptive rules. In order to execute such rules, this system provides an interface of iRODS rule language [13]. iRODS Rule language can be used to define policies and actions in the system. In iRODS Rule language everything is a rule. An example of iRODS rule is described below

```
acPostProcForPut {
     on($objPath like "*.txt") {
         msiDataObjCopy($objPath, "$objPath.copy");
     }
}
```

In this rule, $acPostProcForPut$ is the name of the rule. This rule is automatically triggered when a condition described in $on()$ clause is true for an event. A $\{\}$ followed by a rule condition is executed when the rule is applied. A detailed description of this language is defined in [13].

As you can see, syntactic structure of this language is very similar to a procedural language like C. So, direct translation of policies in natural language to such rules would be very difficult. However, such translation would be doable if we extract the knowledge from natural language in some intermediate declarative language format, which then can be converted to iRODS rules. Thus, this intermediate language would interpret some kind of formalism over natural language. In next section, we will define the syntax of Intermediate Policy Declarative language.

2.2 Syntax of Intermediate Policy Declarative language (IPDL)

For the purpose of this research, we have designed a very simple IPDL language which is a combination of First Order Logic and Events. In order to design syntax of this language, we have categorized iRODS rules in 3 categories: (1) Trigger rules, (2) Validation rules, (3) Direct commands. A formal representation of iRODS policies/rules in IPDL can be any combination of these categories. Each of these rules contain basic symbols: $event$, $condition$. An $event$ symbol is a function symbol of arity n which defines a specific operation in iRODS system. Thus, an event could be a micro-service or a pre-defined procedure already provided by a user. A basic $condition$ symbol is a predicate symbol of arity m which defines a state information, a description of an entity/object in iRODS domain or of the form $m_1 \ \Theta \ m_2$, where Θ is a relational operator ($\leq, \geq, <, >, =$) and $m_1,\ m_2$ are predicate symbol terms. A formal definition of IPDL is described below.

Before defining syntax of IPDL let us first define the signature of this language as $S = (\Omega, \Pi, R, D)$.

- Ω is a set of function symbols f with arity $n \geq 0$, written as f/n. If $n = 0$, then f is called constant symbol.
- Π is a set of predicate symbols p with arity $m \geq 0$, written as p/m.
- R is a set of relation symbols. (eg. $\leq, \geq, <, >, =$)
- D is the domain of iRODS system which includes a set of micro-services, a set of data objects and collections, a set of numbers and mathematical symbols etc.

Now we will define syntactic formula for IPDL language.

$$
\begin{aligned}
< formula > \ ::= \ & < rule > \ | \ \neg < formula > \ | \\
& < formula > \wedge < formula > \ | \\
& < formula > \vee < formula > \ |
\end{aligned}
$$

$$< rule > \ ::= \ < trigger\_rule > \ | \ < validation\_rule > \ | \ < command >$$

$$\begin{aligned} < trigger\_rule > \ ::= \ & when > \ \ < event > \\ & IF \ < condition >; \quad \text{(optional)} \\ & do > \ \ < rule > \end{aligned}$$

$$\begin{aligned} < validation\_rule > \ ::= \ & validation > \ \ < condition > \ | \\ & validation > \ \ < event > \\ & IF \ < condition > \quad \text{(optional)} \end{aligned}$$

$$\begin{aligned} < command > \ ::= \ & command > \ \ < event > \\ & IF \ < condition > \quad \text{(optional)} \end{aligned}$$

$$\begin{aligned} < condition > \ ::= \ & < term > \ | \ \neg < condition > \ | \\ & < condition > \wedge < condition > \ | \\ & < condition > \vee < condition > \ | \end{aligned}$$

$$< event > \ ::= \ f(< term >_1, < term >_2, ..., < term >_n) \ \ , f/n \in \Omega$$

$$< term > \ ::= \ p(< term >_1, < term >_2, ..., < term >_m) \ \ , p/m \in \Pi$$

An illustration of syntax of IPDL with some iRODS policies examples is given in table 2.1.

Table 2.1: Representation of Typed FOL λ-calculus

Policy	IPDL Representation
Print staff experience report	$command > print(report(staff\_experience))$
Check if file is master copy	$validation > master\_copy(file)$
Compare AIP content with AIP template and list all non-compliant files	$command > compare(content(aip), template(aip)) \wedge$ $command > list(files(non_compliant))$
Send report on evaluation of assessment criteria to certifying body	$command > send(report(evaluation(assessment\_criteria)),$ $certifying\_body)$
On ingestion into collection gamma send an email to curator of the collection	$when > ingestion(collection(gamma));$ $do > command > send(email, curator(collection))$
Validate checksum	$validation > checksum$
On ingestion to collection beta make sure that the file type is that of an image file	$when > ingestion(collection(beta));$ $do > validation > file(type) = file(image)$
When resource resc-24 is full migrate files to resource resc-25	$when > full(resource(resc\_24));$ $do > command > migrate(files, resource(resc\_25))$
On insertion of metadata flag1 for files in collection omega perform rule24 and set metadata flag flag2 on success	$when > insertion(metadata(flag1), collection(omega));$ $do > command > perform(rule24) \wedge$ $command > set(metadata(flag2))\ IF\ success$
On access of file from collection if the user is from eu group perform redact1 before delivering file	$when > access(collection(file))$ $IF\ group(user) \wedge name(group, eu);$ $do > command > perform(redact1)\ IF\ \neg deliver(file))$
Create AIP template from SIP template	$command > create(template(aip), template(sip))$
Move Leica folder to Trash	$command > move(folder(leica), trash)$
Check if iota is collection	$validation > collection(iota)$

Chapter 3

NL2KR v.2 System

3.1 Shortcomings of NL2KR v.1 system

We have already discussed about existing NL2KR v.1 system in chapter 1. As mentioned previously, v.1 is a very basic implementation of NL2KR system [1] aimed to verify correctness of some concepts in the theory. However there are various shortcomings in the system. Several components of the v.1 system needed revamping. In this section, we will discuss about deficiencies in the NL2KR v.1 system.

- *Integrated NL2KR-L and NL2KR-T components* - A major drawback of the NL2KR v.1 system is integrated NL2KR-L and NL2KR-T components i.e. there are no separate subsystems for these components. So, a user has to run the complete system (training and translation) in order to translate a sentence to a formal representation.

- *CCG Parser* - The NL2KR v.1 system has an inbuilt CCG parser which is incapable of generating all possible combinations of CCG parse trees given CCG categories of words. Hence, some of the good parse trees which could help in learning good semantics of words can be eliminated.

- *Inverse-λ* - In the NL2KR v.1 system, a basic version of Inverse-λ component was implemented which did not cover all the cases of Inverse-λ definition. This component was able to compute missing λ-expression of a child given very simple λ-expressions of it's parent and sibling. For complex λ-expressions, it was not able to compute any result. Hence the system performance was greatly affected.

- *Generalization* - Even though v.1 of the NL2KR system uses Generalization on demand while learning new semantics of words in a parse tree of a sentence, during learning phase, it still produces a large amount of new unnecessary semantics which could have negative impact on performance of the system. Also, while generalizing lexicon, a default weight is assigned to each lexical entry which could also affect the translation process.

Apart from this, the generalization component does not recognize singular and plural words. The details about these drawbacks and their solutions is explained in section 3.6. Also, some configuration needed to enhance this component's performance is also described in details in section 3.8.

- *Learning Algorithm* - Current learning algorithm in v.1 of the system, was designed to learn only missing semantics of words in a sentence using Inverse-λ and generalization techniques. However, a word can have multiple semantics given a CCG category. There could be possibility that a semantics of a word with a given CCG category is learned by parsing one sentence, might not be applicable to construct meaning of another sentence. For eg. consider two iRODS policy statements 'Update collection' (Translation: $command > update(collection)$) and 'Delete iota collection' (Translation: $command > delete(collection(iota))$). In both of these sentences, 'collection' has category N. However, a meaning of 'collection' in the first sentence should be $collection$ while in the second sentence, it should be $\lambda x.x@\lambda y.collection(y)$. With existing learning algorithm, the system would learn a meaning of 'collection' by parsing first sentence. Now while parsing second sentence, it would use the learned meaning of 'collection' i.e. $collection$ as it is not missing even though is not compatible with our formal representation of the sentence. So, the system won't learn new meaning of 'collection' again.

3.2 NL2KR v.2 System Architecture

An overview of the NL2KR v.2 system architecture is described in figure 3.1. In next sections, we will elaborate on improvements made in most of the components of the system.

3.3 Implementation of CCG Parser using ASP+Lua

As mentioned in previous section, an existing internal CCG parser of NL2KR v.1 system did not generate all possible combinations of CCG parse trees given CCG categories of words because of which some of the good parse trees which could help in learning good semantics of words were eliminated. Most of the times, existing CCG parser did not return any parse tree

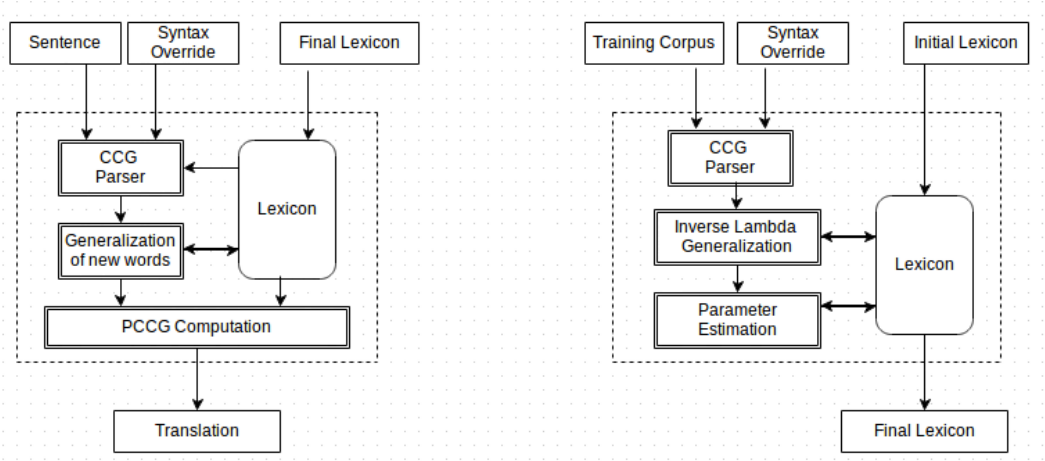


Figure 3.1: Overview of NL2KR v.2 system

even if given CCG categories of words could produce a parse tree for a sentence. This parser is implemented in Java and uses Stack data structure for implementation. However, with this stack data structure, the parser was able to compose a parse tree based on adjacent word's CCG category only (did not consider adjacent phrases) which was a major flaw in implementation.

In NL2KR v.2 system, we have implemented an inbuilt CCG parser using ASP and Lua. We know that obtaining all possible combinations of CCG parse trees given CCG categories of words in a sentence is a search problem where we would need to search for all possible compositions of adjacent words or phrases such that a root of a parse tree contains all words in a sentence. ASP is an effective declarative programming language which is efficient in solving such problems. However, in order to compose CCG category of a node from its children we needed a functionality that is efficiently handled by Lua.

Before discussing about our implementation in ASP we will first present our ASP representation of a CCG parse tree. We say that each node in a parse tree is a token of the format $nl2kr\_token(TokenId,\ Word/Phrase,\ CCG,\ Position)$ where $TokenId$ is a unique token, $Word/Phrase$ is a word/phrase which is a part of a sentence, CCG is a CCG category of a word/phrase, and $Position$ is a position of a word/phrase in a sentence.

As a CCG parse tree is a binary tree, we represent a left child link between nodes in ASP as follows: $nl2kr\_child\_left(Parent\_TID, Child\_TID)$ where $Parent\_TID$ is a unique token id of a parent and $Child\_TID$ is a unique token id of a child. Similarly we represent a right child link between two nodes as $nl2kr\_child\_right(Parent\_TID, Child\_TID)$.

We assume that a root node of a parse tree will always have category S. Hence, our parser will find a valid parse tree if there exists CCG categories of words/phrases in a sentence such that their compositions will result in a sentence with CCG category S. We represent a valid root node as $nl2kr\_valid\_rootNode(TokenId)$ where TokenId is a token id of a node.

Example 1 *A parse tree of a sentence "Every boxer walks" is represented in our ASP format as follows*

$nl2kr\_token(t1, \text{“}Every\text{”}, \text{“}(S/(S\backslash NP))/NP\text{”}, 1).$

$nl2kr\_token(t2, \text{“}boxer\text{”}, \text{“}NP\text{”}, 2).$

$nl2kr\_token(t3, \text{“}walks\text{”}, \text{“}S\backslash NP\text{”}, 3).$

$nl2kr\_token(t4, \text{“}Every\ boxer\text{”}, \text{“}S/(S\backslash NP)\text{”}, 1).$

$nl2kr\_child\_left(t4, t1).$

$nl2kr\_child\_right(t4, t2).$

$nl2kr\_token(t5, \text{“}Every\ boxer\ walks'\text{”}, \text{“}S\text{”}, 2).$

$nl2kr\_child\_left(t5, t4).$

$nl2kr\_child\_right(t5, t3).$

$nl2kr\_valid\_rootNode(t5).$

Now we will present Lua methods being used for implementation of the parser.

- $function\ token\_id()$

  This method will generate a new token id

  Usage in ASP: $ID := @token\_id()$

- $function\ current\_id()$

  This method returns a current id of a token which is recently generated by method

$token\_id()$

Usage in ASP: $ID := current\_id()$

- $function\ combine(ccg1, ccg2)$

  This method combines CCG category $ccg1$ with $ccg2$. For eg. a result of combination of S/NP with NP will be S.

  Usage in ASP: $CCG := @combine(CCG1, CCG2)$

- $function\ canCombine(ccg1, ccg2)$

  This method check if a CCG category $ccg1$ can be combined with $ccg2$. The method return 1 if it is a forward composition, return 2 if it is backward composition and return 0 if $ccg1$ cannot be combined with $ccg2$

  Usage in ASP: $Result := @canCombine(CCG1, CCG2)$

- $function\ belongs\_to(s, t)$

  This method check if a word/phrase t belongs to a sentence s.

  Usage in ASP: $Result := @belongs\_to("Every\ boxer", "boxer")$

- $function\ append(a, b)$

  This method appends string b to a

  Usage in ASP: $Result := append("Every", "boxer")$

Our parser takes a sentence and words/phrases in a sentence as an input in the format of $nl2kr\_token(TokenId,\ Word/Phrase,\ CCG,\ Position)$ illustrated before and considers them as leaf nodes. For eg. for the sentence “Ever boxer walks", the input will be $sentence(\text{“}Every\ boxer\ walks\text{”})$.

$nl2kr\_token(t1, \text{“}Every\text{”}, \text{“}(S/(S\backslash NP))/NP\text{”}, 1)$.

$nl2kr\_token(t2, \text{“}boxer\text{”}, \text{“}NP\text{”}, 2)$.

$nl2kr\_token(t3, \text{“}walks\text{”}, \text{“}S\backslash NP\text{”}, 3)$.

Now we will describe ASP rules which take an input described above and generate parse trees.

We first define possible forward and backward applications between tokens considering their positions in a sentence. For eg. one possible forward application would be between $Every\ (\ (S/(S\backslash NP))/NP\ )$ and $boxer\ (NP)$. Below are the rules which define possible forward and backward applications.

```
valid_token(W, C, P) :- nl2kr_token(T, W, C, P), sentence(S),
                        @belongs_to(S, A) == true.

possible_backward_appl(T1, T2) :- nl2kr_token(T1, W1, C1, P1),
                                  nl2kr_token(T2, W2, C2, P2),
                                  X:= @canCombine(C1,C2), X == 1, P1 > P2,
                                  valid_token(W1, C1, P1),
                                  valid_token(W2, C2, P2).

possible_forward_appl(T1, T2) :- nl2kr_token(T1, W1, C1, P1),
                                 nl2kr_token(T2, W2, C2, P2),
                                 X:= @canCombine(C1, C2), X == 2, P1 < P2,
                                 valid_token(W1, C1, P1),
                                 valid_token(W2, C2, P2).
```

Using above possible forward and backward applications between two nodes $n1$ and $n2$, we create intermediate nodes and their left and right child links. Below mentioned rules specify this.

```
%forward application
3{ nl2kr_token(@token_id(), @append(A,B), @combine(C1, C2), P1),
```

```
   nl2kr_child_left(@current_tID(), T1),
   nl2kr_child_right(@current_tID(), T2)
 }3
:- nl2kr_token(T1, A, C1, P1), nl2kr_token(T2, B, C2, P2),
   possible_forward_appl(T1, T2), A != B, C1 != C2, P1 != P2.

%backward application
3{ nl2kr_token(@token_id(), @append(B, A), @combine(C1, C2), P2),
   nl2kr_child_left(@current_tID(), T2),
   nl2kr_child_right(@current_tID(), T1) }3
:- nl2kr_token(T1, A, C1, P1), nl2kr_token(T2, B, C2, P2),
   possible_backward_appl(T1, T2), A != B, C1 != C2, P1 != P2.
```

Finally, we say that a node is a valid root node if it's CCG category is S and it's phrase is a sentence. The number of parse trees generated for a sentence is the number of valid root nodes.

```
nl2kr_valid_rootNode(T) :- nl2kr_token(T,A, "S", P), sentence(S), A = S.
```

3.4 Integration of ASPccgTk CCG Parser

One of the drawbacks of using inbuilt CCG parser is, we have to provide CCG categories of all the words in a sentence. Using inbuilt parser will not scale our system for larger training and test data because for each sentence in training and test data set, we will need to provide CCG categories of all the words. Hence, in order make our system scalable we will need to use the existing CCG parser. $C\&C\ parser$ ([11] and [12]) uses the grammar from CCGBank and it efficiently parses a sentence and returns a most probable parse tree. However, the parse tree generated by $C\&C\ parser$ might not be a good parse tree for learning new semantics of words corresponding to a given target language. Also, most of the times we want to specify a specific CCG category for some words to increase performance of the system so that we can learn

good semantics of such words. As $C\&C\ parser's$ model is already trained on the CCGBank, it is very difficult to override CCG categories of words during parsing process.

We need an efficient CCG parser which uses existing CCG grammar resources like CCGBank, PennTreeBank and provides functionality of overriding CCG categories. $ASPccgTk$ CCG Parser [14] considers CCG parsing as a planning problem and uses ASP to solve this problem. $ASPccgTk$ CCG Parser provides an option of using our own hand-crafted grammar and using a grammar generated by $C\&C\ Supertagger$ [15]. One of the main advantage of using this parser is, we can tweak the source code of the parser easily to fit to our requirements which would reduce post-processing task to a great extent. Hence, we have integrated this parser with our NL2KR v.2 system. We will now elaborate on how we have integrated it with our system and what changes we have made in the parser to easily integrate it with our system.

CCG Parsing

We have already presented our ASP representation of a parse tree which our system uses, in the previous section. However, an output of the $ASPccgTk$ CCG Parser is in a grid format. So, we need to convert this output to our ASP format. We have a wrapper program written in ASP which takes a stable model output of the $ASPccgTk$ CCG Parser as input and frames it to our own format. A stable model output of the $ASPccgTk$ CCG Parser contains following predicates which would help us create a parse tree in our format:

- $word\_at(Word, Position)$ where $Word$ is a word in a sentence, $Position$ is a position of a word in a sentence.
- $reachable(Position1, Position2, CCG)$ which specifies that a word at $Position1$ is reachable by another word at $Positions2$ through CCG.

Following rules creates possible parse tree tokens and their corresponding left and right children.

```
2{ nl2kr_ptoken(@token_id(), W, @ccg(C), -1),
   nl2kr_node(@current_tID(), P,P) }2  :- word_at(W, P), reachable(P,P, C).
```

```
4{ nl2kr_ptoken(@token_id(), @append(W1,W2), @ccg(Z), -1),
   nl2kr_node(@current_tID(), X, Y),
   nl2kr_child_left(@current_tID(), T1),
   nl2kr_child_right(@current_tID(), T2) }4
:- nl2kr_node(T1, X1, Y),
   nl2kr_node(T2, X, Y1),
   (Y1-X1) == 1,
   reachable(X,Y,Z), T1 != T2,
   nl2kr_ptoken(T1, W1, C1, P1),
   nl2kr_ptoken(T2, W2, C2, P2).
```

First rule specifies that, if a word is reachable to itself then it is a possible basic token i.e. a leaf node. Predicate $nl2kr\_ptoken$ has the same type of arguments as that of $nl2kr\_token$, however, here we set the $Position$ argument as -1 because this information is already provided by $word\_at$ predicate. Predicate $nl2kr\_node$ will help us map the position reachability information of words/phrases in a parse tree output of the $ASPccgTk$ CCG Parser to our parse tree so that we can correctly connect a token with it's left and right child. Methods $token\_id()$, $append(W1, W2)$, $current\_id()$ performs the same functionality as described in the previous section. $ccg(C)$ method converts the $ASPccgTk$ CCG Parser's CCG category i.e. C format to our format. $ASPccgTk$ CCG Parser represents a CCG category in functional representation Eg. Category "S/NP" is represented as $rfunc(`S', `NP')$. ccg method converts $rfunc(`S', `NP')$ back to normal format i.e. "S/NP".

Second rule states that, if two nodes are reachable through category Z and their positions are adjacent to each other then create an intermediate node having category Z.

$ASPccgTk$ CCG Parser uses a special category $conj$ for conjunctions and disjunctions. For eg. Consider a phrase 'tea and coffee'. CCG category for 'tea' is NP, for 'coffee'

NP and for 'and' is $conj$ and a CCG category for the phrase is NP. But our system does not understand $conj$ category which binds us to convert it into a normal CCG category. But looking at all CCG categories of words and the phrase we can easily guess that the CCG category of 'and' will be $(NP\backslash NP)/NP$. Following rule defines this conversion.

```
4{ nl2kr_token(TRL, WRL, @catReplace(CR, "conj", @leftappend(C,CL)), PRL),
   nl2kr_token(TR, WR, @catReplace(CR, CR, @leftappend(C, CL)), PR),
   invalid_nl2kr_token(TR, WR, CR, PR),
   invalid_nl2kr_token(TRL, WRL, "conj", PRL)
 }4
    :- nl2kr_ptoken(T, W, C, P),
    nl2kr_child_left(T, TL), nl2kr_ptoken(TL, WL, CL, PL),
    nl2kr_child_right(T,TR), nl2kr_ptoken(TR, WR, CR, PR),
    nl2kr_child_left(TR, TRL), nl2kr_ptoken(TRL, WRL, "conj", PRL),
    nl2kr_child_right(TR,TRR), nl2kr_ptoken(TRR, WRR, CRR, PRR),
    @contains(CR, "conj") == true,
    T != TRL, T!= TRR.
```

In the above rule, some new Lua methods are used which are not described earlier. $catReplace(CCG, Pattern, Value)$ method replaces a $Pattern$ in CCG with a $Value$. $leftappend(CCG1, CCG2)$ appends a $CCG2$ category to left of CCG1 i.e. it creates a backward application format. $contains(CCG, Pattern)$ checks if a $Pattern$ is present in the CCG category.

Next rule creates valid $nl2kr\_token$ predicates from possible tokens if they are not invalid.

```
nl2kr_token(T,W,C,P) :- nl2kr_ptoken(T,W,C,P),
                        not invalid_nl2kr_token(T,W,C,P).
```

Again, we need to know a valid root node of a parse tree, which can be easily found using below rule.

```
nl2kr_valid_rootNode(T) :- nl2kr_token(T,A, C, P), sentence(S), A = S.
```

Currently, our system understands parse trees whose nodes are connected using forward and backward applications. So, we eliminate parse trees in which nodes are reachable through other combinators like forward cross composition, backward cross composition among others. These combinators are given in $ASPccgTk$ CCG parser.

```
combinator(rule_fwd_comp;rule_bwd_comp).
combinator(rule_bwd_x_comp; rule_bwd_x_subst).
invalid_tree :- occurs_grid(Z, A,B,C,D,E,F,G), combinator(Z).
:- invalid_tree.
```

Including semantics in Parse Trees

We know that a word with a given CCG category can have multiple meanings. For eg. again consider a sentence 'Every boxer walks'. If we have 2 different meanings for words 'Every' and 'boxer' with given CCG category, then total possible parse trees will be 4 where each parse tree will have unique set of meanings of words. This can be easily implemented in ASP. So instead of post-processing each parse tree and creating duplicate copies of parse trees with unique set of meanings of words, we can easily extend the ASP parse tree wrapper program described in previous section, to include this functionality.

Now we will describe rules which will help use generate multiple parse trees with unique set of meanings. We have already mentioned that this wrapper program takes a stable model output of the $ASPccgTk$ CCG parser as input. We will also take semantics of words in a sentence with their corresponding CCG categories, as input. An ASP format for this input is $sem(Word, CCG, Semantics)$.

First, we provide a unique semantics id for each semantics input.

```
1 { semantics(@sem_id(), T, W, Z, SS) } 1:- sem(W, CCG, SS),
                                            nl2kr_token(T, W, Z,P),
                                            Z := @process(CCG).
```

Here, $process(CCG)$ is a lua method which trims a CCG category if necessary.

Following code generates all possible combination of semantics i.e. a stable model is generated for each unique combination.

```
% A semantics is either selected or not selected in a stable model
selected(T) | not_selected(T) :- semantics(T ,TT, W, CCG, SS).

% remove all stable models in which a given node token
% has at-least 2 semantic meanings
:- selected(T1) , selected(T2), semantics(T1, TT, W, C, S1),
   semantics(T2, TT, W, C, S2), T1 != T2.

% These rules specify that each stable model should have at-most
% one semantics for all the words whose semantics is taken as input
words(W) :- semantics(T, TT, W, CCG, M).
hasVisited(W) :- words(W), selected(T), semantics(T,TT, W,C,SS).
:- words(W), not hasVisited(W).

% assign a selected semantics to a token
nl2kr_semantics(TT, expression(SS)) :- selected(T),
                                       semantics(T,TT, W,C,SS).
```

3.5 Implementation of Inverse-λ Algorithm (Java)

Inverse-λ operators defined in [2] resemble a typical search problem. We have implemented Inverse-λ operator definitions described in Marcos thesis [2] in Java programming language, however with some restrictions for simplicity and efficiency of the algorithm. Typically, in definition 3 of $Inverse_L$ and $Inverse_R$ operators, we look for a pattern of the form $J^1(J^1_1, ..., J^1_m), ..., J^n(J^n_1, .., J^n_m))$, in H. We assume that J^1, J^2, $, ...,$ J^n are strict predicates and $(J^i_1, ..., J^i_m)$ are arguments of J^i where each J^i_j can be a predicate of the form $J^j(J^j_1, ..., J^j_m)$. For eg. in $\lambda x.(plane(x) \wedge takes(x, Y))$ valid J^i's are $plane(x)$, $takes(x, Y)$ and $(plane(x) \wedge takes(x, Y))$. With this restriction we limit the possible set of sub terms in H and hence help in reducing the complexity of finding the sub-terms in H which match with G.

3.6 Implementation of Generalization Algorithm

As mentioned in section 3.1, existing generalization algorithm does not recognize a singular and plural word. We have modified current Generalization algorithm to consider singular and plural words. We have modified the $IDENTIFY$ method which identifies a part of all the lexicon entries of a word such that word can have singular or plural form in the semantics.

As described in chapter 1's section 1.1, a default weight of 0.01 is assigned to each new lexical entry learned. Although, when we perform generalization on demand during learning phase by parsing training data parse trees, a default weight won't affect new translations because during the parameter estimation phase weight of these new lexical entries learnt will be estimated. However, in the NL2KR-T component when generalization of words in a sentence with missing meanings is done, a default weights assigned to new lexical entries will affect translation of a sentence. We know that using generalization technique, we might get different meanings of a new word based on it's syntactic category. Moreover, semantics of a new word can be learned from different lexical entries in the lexicon having same syntactic category as that of new word but with different semantics.

For eg. Consider two lexical entries from lexicon $(takes,\ (S\backslash NP)/NP)$, $\lambda w.\lambda z.has(w, takes, z)$ and $(loves,\ (S\backslash NP)/NP,\ \lambda w.\lambda z.loves(w, z)$. We want to learn meaning of a new word $likes$ having syntactic category $(S\backslash NP)/NP)$. Using generalization technique, semantics of $likes$ learned would be $\lambda w.\lambda z.has(w, likes, z)$ and $\lambda w.\lambda z.likes(w, z)$ and a default weight 0.01 will be assigned to each of the lexical entries corresponding to these semantics. Now if we want to translate a sentence 'Vincent likes Mia', we can get two possible translations for this sentence given semantics of 'Vincent' as $vincent$, 'Mia' as mia and 'likes' will have semantics learned through generalization. Translations are as follows:

$$has(vincent, likes, food)$$
$$likes(vincent, food)$$

We already know that NL2KR-T system uses PCCG to compute a translation of a sentence. The idea behind PCCG is that given a sentence S, its translation M can be obtained by finding M which maximizes the probability function $P(M|S; \Theta)$. Now given fixed weights for lexical entries of 'Vicent' and 'Mia', we will obtain 2 different translations having same weight because there are two different semantics of 'likes' having same default weight. So, the NL2KR-T system will apparently randomly pick one of these translations which might not be the correct one. In order to resolve this problem to some extent, we assign a weight to a new lexical entry as that of the one from which it is learned.

However, a particular semantics of a new word can be learned from multiple words which have same CCG category. For eg. semantics of 'likes' - $\lambda w.\lambda z.likes(w, z)$ can be learned from words 'loves' - $\lambda w.\lambda z.loves(w, z)$ and 'eats'- $\lambda w.\lambda z.eats(w, z)$. So in such cases we assign a weight to the new lexical entry as that of the one which has maximum weight.

Algorithm for Generalization is as follows:

Algorithm 3 $Generalize_D(\Lambda, \alpha)$

for $l_i \in \Lambda$ **do**
  if $l_i(category) = \alpha(category)$ **then**
    $I = IDENTIFY(l_i(word), l_i(semantics))$
    $S = REPLACE(l_i(semantics), I, \alpha(word))$
    Update weight W of S
    $\Lambda = \Lambda \cup (\alpha(word), \alpha(category), S, W)$
  end if
end for

3.7 Implementation of Learning Algorithm

We have already discussed the shortcomings of the Learning algorithm in NL2KR v.1 system. To overcome such problems, we will describe a better lexicon learning algorithm for our NL2KR-L component, in this section. This algorithm is mainly designed by Nyugen Vo with my assistance.

Now we will first mention some of the keywords and properties used in this algorithm.

- A node A in a parse tree can have two semantics: (1) expected semantics, (2) current semantics. Expected semantics is obtained using a training data in the learning process. Current semantics is obtained from an initial lexicon. An intermediate node's current semantics will be obtained from its descendants through λ-application.
- A has current semantics if and only if all of its children have current semantics.
- If node A has an expected semantics then it's parent will also have an expected semantics.

Algorithm is as follows:

Algorithm 4 NL2KR-L Lexicon Learning Algorithm

function TRAVERSEPARSETREE(Node root)
push root in the Stack S
while S is not empty **do**
Node A = Pop node from S
Push all the children of A in stack S
if (A does not have current semantics) || (A's expected $\neq$ null $\&\&$ it's current semantics $\neq$ expected semantics) **then**
LEARN(A)
end if
end while
end function
function LEARN(Node A)
if A is leaf node and has expected semantics **then**
Update the lexicon using the expected semantics of the node
end if
if only one child of A i.e. B does not have current semantics **then**
if B is the only child of A **then**
Learned semantics of B = expected semantics of A
else if If B's sibling i.e. C has current semantics **then**
Learned semantics of B = Inverse-λ(A's expected semantics, C's current semantics) if A has expected semantics
end if
Set the learned semantics of B as its expected semantics
end if
if both the children of A have current semantics **then**
Set the expected semantics of one child as its current and use the expected semantics of A and this child to compute the semantics of other one using Inverse-λ if A's expected semantics exists and visa versa.
end if
if any child of A is leaf and it still does not have expected semantics **then**
Use $Generalization_D$ technique to learn the expected semantics of children and update lexicon
end if
end function

3.8 Bug Fixes and New Features

There were some bugs in the existing NL2KR v.1 system. In the v.2 version of the NL2KR system, we have fixed these bugs and also added some new features in the system to improve performance of the system and increase it's usability.

- The source code in v.1 is not well organized and code is not easily readable by developers. In version v.2 of the system, we have tried to modularize the code as much as possible by maintaining code for a component in one package and it's subcomponents in corresponding sub-packages. Also, to make the code readable and understandable to other developers we have provided comments for each procedure.

- We have already mentioned in chapter 1 that the v.1 system does not have separate sub-systems for NL2KR-L and NL2KR-T components which forces a user to every time train the system before using it for translation. In v.2, we have separated these two components. A user can train the system anytime by running NL2KR-L sub-system. After training the system, a final updated lexicon is saved. A user then, can use the NL2KR-T sub-system any time to translate a sentence to a formal representation using the final lexicon generated by NL2KR-L sub-system.

- In v.1, the lexicon has duplicate entries of lexical entity because of which the parameter estimation component was not estimating a correct weight for a specific lexical entity. We have fixed this issue in v.2. Now, the lexicon will always have unique entry of a lexical entity.

- During parameter estimation phase in v.1, computation of feature vector for the parse trees of a sentence was taking huge amount of time because of exponential increase in number of parse trees. In order to reduce computation time, we have implemented a cache to save feature vectors of all the parse trees of a sentence so that they can be reused in subsequent iterations of parameter estimation. Also, to reduce the number of iterations to compute a feature vector we have implemented a map which will save an

index positions of features in lexicon. So, we don't have to iterate over complete set of lexicon features to get a feature vector table for a parse tree. This improved performance of the system by approximately 8 times.

- Sometimes, NL2KR-L component does not learn semantics of all the words in a training data set because of various factors like failure of Inverse-λ or generalization to learn good semantics, or bad CCG parse tree of a sentence. Now during the parameter estimation phase, a feature vector table is created for each parse tree of a sentence which is later used for estimating parameters of each lexical entry. If semantics of all the words in a sentence doesn't comply to a formal representation, then a root node of a parse tree of a sentence will be null. It is useless to compute feature vector for such parse tree in a table because it won't help in estimating parameters. Hence we have put a condition of semantics not being null while creating feature vector table for a sentence. This fix has removed an exception 'NullPointerException' occurred during parameter estimation.

- Although, there are positive aspects of generalization technique, there are some drawbacks also. It can produce a large amount of new unnecessary semantics which can have negative impact on performance of the system. In order to limit these unnecessary semantics, we have introduced some configuration in config.properties file. Following are the configurations:

    1. $GENERALIZATION\_D\_EXCLIST$ - While learning a new semantics of words in a parse tree of a sentence during learning phase, generalization on demand is used. In order to limit generalization of some categories in generalization on demand, we use $GENERALIZATION\_D\_EXCLIST$ exclude list. It contains list of CCG categories which we want to exclude.

    2. $LEX\_GENERALIZATION\_EXCLIST$ - In the end of lexicon learning phase, generalization is used over complete lexicon. In order to limit generalization of some categories in generalization of lexicon, we use $LEX\_GENERALIZATION\_EXCLIST$ exclude list. It contains list of CCG categories which we want to exclude.

3. $GENERALIZATION\_D\_PREPOSITIONLIST$ - Most of the times, some propositions in English natural language have same CCG categories and similar semantic structures too. But, usually semantics of prepositions does not contain an exact word. Most of the times, it is a lambda expression containing only lambda variables. To improve the system performance and to limit the initial lexicon, we have provided this configuration in config.properties. $GENERALIZATION\_D\_PREPOSITIONLIST$ contains a list of prepositions which we want to generalize from each other. For eg. for iRODS domain, we create a preposition list of on, of, by, to among others.

4. $GENERALIZATION\_D\_EXCWORD\_LIST$ - If you want to exclude generalization of some words regardless of their CCG categories, then we put such words in this exclude list.

- In the previous sections, we have talked about CCG parsers (Existing ASPccgTk parser [14] and inbuilt CCG parser). In order to ease an access of these parsers to the system, we have provided a configuration in config.properties file which would allow us to use either of them in our system. $CCGPARSER$ is a configuration which we need to set to $ASPccgTk$ for ASPccgTk parser and to $SimpleEnParser$ if we want to use inbuilt parser.

Comparison between NL2KR v.1 and NL2KR v.2 system

In this section we will highlight major differences between NL2KR v.1 and v.2 systems as shown in table 3.1.

Table 3.1: Comparison between NL2KR v.1 and v.2 system

NL2KR v.1 System	NL2KR v.2 System
NL2KR-L and NL2KR-T components are tightly coupled together in one component.	Separate systems for NL2KR-L and NL2KR-T components.
Inbuilt CCG parser with restricted implementation.	Inbuilt CCG parser in ASP which generates all possible parse tree derivations and integrated ASPccgTk parser.
Limited implementation of Inverse-λ algorithm. It does not cover all the cases.	Java implementation covers all the cases with one restriction. ASP implementation is full-fledged.
Basic implementation of Generalization algorithm.	New improvised generalization algorithm.
Lexicon Learning Algorithm was designed to learn only missing meanings of words for given CCG category.	New improvised Lexicon Learning Algorithm capable of learning multiple meanings of words for given CCG category.
Limited system configuration.	Included many configuration of different components in the system for better performance.
Basic implementation of Parameter Estimation Algorithm. Complexity to compute the feature vector for each parse tree is exponential.	Improved the Parameter Estimation Algorithm by implementing the cache to handle feature vector computation. Improved the performance of the system by approximately 8 times.
Learning of words beyond training data set was done in the lexicon generalization phase during NL2KR-L learning.	Added the generalization component in NL2KR-T system so that missing meanings of new encountered words will be learned during translation phase itself.
Final lexicon generated has duplicate lexical entries which affects the parameter estimation phase resulting in incorrect computation of weights.	Final generated lexicon has unique lexical entries.

Chapter 4

λ-calculus Operations and Inverse-λ in ASP+Lua

In Marcos thesis [2], a detailed study of formal languages in terms of λ-calculus formalism is described and Inverse-λ technique is mentioned in terms of these formalism. In this research, we would like to go a step further. We would like to represent a typed λ-calculus in ASP. One of the major reasons for this approach is, ASP is easily extensible to new knowledge which would be useful when we want to use world knowledge for correct translation of natural language to a formal representation using λ-calculus. For eg. while constructing meaning or formal semantics of a sentence from meanings of words/phrases in it, we might come across multiple meanings of a given word which are applicable in a sentence's meaning construction using λ-application. So, we will need to pick a correct meaning of a word i.e. correct sense in order to correctly translate a sentence to its formal representation. In order to do it, we will need world knowledge to disambiguate senses of words which could be easily represented in ASP. Also, we can easily write some generic rules in ASP which would help in disambiguating the senses.

We also know from the definition of Inverse-λ algorithm [2], that it is a typical search problem in space where we look for a pattern F such that H=F@G or H=G@F. This pattern is nothing but a λ-calculus expression. Thus, with our succinct representation of λ-calculus expression in ASP, we can easily implement Inverse-λ definition by writing few rules in ASP which is an efficient declarative programming paradigm to solve such problems.

In this chapter, we will talk about a generic representation of λ-calculus expressions in ASP and Inverse-λ algorithm in ASP.

4.1 Representation of λ-calculus expression in ASP

In order to implement λ-operations in ASP, we need a succinct representation of λ-calculus formulas in ASP. In this section, we will introduce a generic representation of λ-calculus in ASP in accordance with the "Simply typed λ-calculus" in [16].

λ-calculus signature

The signature of λ-calculus in ASP is defined below

1. A lambda operator 'λ' represented in ASP as a predicate symbol 'l'.
2. A lambda application '@' represented in ASP as a predicate symbol 'a'.
3. A parenthesis (,)
4. An inifinte set of variables r_n for each natural number n

Representation of λ-Abstraction in ASP

λx. E is a λ-abstraction where x is a bound λ variable and E is a λ-expression. We represent it in ASP as follows

$$l(x, E)$$

A complex λ-expression having multiple variables can be represented in ASP in nested structure format. λv1. λv2... λvn. E can be represented in ASP format as follows

$$l(v1, l(v2, ...l(vn, E)))$$

Representation of λ-Application in ASP

M@N is a λ-application where M, N are λ-expressions. ASP representation of M@N is

$$a(M, N)$$

Similar to a λ-abstraction, a complex λ-application term can be easily represented in ASP in nested structure format.

Table 4.1: Representation of Typed FOL λ-calculus

Category	FOL Symbol	ASP Representation
Quantifiers	$\forall$	$forall$
	$\exists$	$exists$
Connectives	$\wedge$	and
	$\vee$	or
	$\rightarrow$	$imply$
Negation	$\neg$	neg
Equality symbol	$=$	$equal$
Function symbol with given arity n	$function(a1, .., an)$	$f(function, a1, ..., an)$

Table 4.2: Examples of Typed FOL λ-calculus

FOL	ASP Representation
$\neg x$	$f(neg, x)$
$x \bigwedge y$	$f(and, x, y)$
xVy	$f(or, x, y)$
$x \rightarrow y$	$f(imply, x, y)$
$x = y$	$f(equal, x, y)$
$loves(x, y)$	$f(loves, x, y)$
$\lambda x.woman(x)$	$l(x, f(woman, x))$
$\forall x.(x \rightarrow h(x))$	$f(forall, x, f(imply, x, f(h, x))$
$\lambda y.\forall x.(x@z \rightarrow y@z)$	$l(y, f(forall, x, f(imply, a(x, z), a(y, z))))$

(E1@E2)@E3 can be represented in ASP as follows

$$a(a(E1, E2), E3)$$

Representation of FOL-λ-calculus in ASP

In Marcos research [2], Typed First-Order-Logic λ-calculus language is introduced. In this section, we talk about representation of this language in ASP. We consider that each FOL expression is of the format $f(expression)$. For eg. FOL expression $a \vee b$ can be represented in ASP as $f(or, a, b)$. An ASP representation of FOL terms/symbols is described in table 4.1. Also, an illustration of examples of typed FOL λ-calculus is given in table 4.2.

Table 4.3: Representation of Typed ASP-λ-calculus

Category	ASP Symbol	Our ASP Representation
Connectives	$\leftarrow$	$imply$
	,	and
	or	or
	|	or
	.	$concat$
Negation	not	$dnot$ (default negation)
	$\neg$	$cnot$ (classical negation)
Equality symbol	$=$	$equal$
Predicate symbol with given arity n	$function(a1, .., an)$	$f(function, a1, ..., an)$

Table 4.4: Examples of Typed FOL λ-calculus

ASP	Our ASP Representation
$a \leftarrow b.$	$f(imply, b, a)$
$a \leftarrow b, c.$	$f(imply, f(and, b, c), a)$
$has(x, property, red).$ $has(x, instance, color).$	$f(concat, f(has, x, property, red),$ $f(has, x, instance, color))$
$\lambda x.(h(a) \leftarrow x@b)$	$l(x, f(imply, a(x, b), f(h, a)))$
$\leftarrow woman(vincent).$	$f(imply, f(woman, vincent), null)$
$\lambda x.\neg\ x@z \leftarrow not\ \ x@z$	$l(x, f(imply, f(dnot, a(x, z)),$ $f(cnot, a(x, z))))$

Representation of ASP-λ-calculus in ASP

To represent ASP-λ-calculus in ASP we follow the same convention mentioned in previous section for FOL-λ-calculus. We consider each ASP rule is of the format $f(rule)$. A representation of ASP terms/symbols is defined in table 4.3 and an illustration of examples of typed ASP-λ-calculus is given in table 4.4.

Special function symbols: concat, null

In our representation, we have introduced a special symbol concat for a ' . ' (dot) operator in ASP. With this symbol, we can represent a complete ASP program as a λ-calculus function.For eg. $a \leftarrow b.\ b \leftarrow c.$ can be represented in our format as follows

$$f(concat, f(imply, b, a), f(imply, c, b))$$

Similarly, we have introduced a special symbol $null$ for the cases where head of an ASP rule is empty. For eg. $\leftarrow b.$ can be represented in our format as $f(imply, b, null)$

Set representation of and, or, concat in ASP

In languages like FOL, ASP there are operators like $\wedge, \vee,$ ',' which connects the sub terms. Eg. $x \vee y \vee z$ which is equivalent to $(x \vee y) \vee z,\ x \vee (y \vee z),\ x \vee z \vee y$ .... Here, (x, y, z) is typically a set where x, y and z are connected to each other with disjunction operator.

Initially our ASP representation for such formulas was

- $(x \vee y) \vee z$ - $f(or, f(or, x, y), z)$
- $x \vee (y \vee z)$ - $f(or, x, f(or, y, z))$

We represented a series of conjunction, disjunction operators as a nested function. One of the drawbacks of this ASP representation was difficulty to get all possible sub-terms of such expressions. For the above expression, total possible sub-terms are 7. Also, it is difficult to perform λ-calculus operations like substitution, replacing one pattern with another, checking equivalence, if we have such nested structure representation of conjunction and disjunction operators.

In order to avoid these complications, we consider conjunction, disjunction operators connecting terms belong to a set. So, the expression $x \vee y \vee z$ is a disjunction set of x, y, z items. Thus our new ASP representation of this expression will be

$x \vee y \vee z$ - $f(or, x, y, z)$

λ-calculus Operations in ASP

In the previous sections, we have introduced a succinct representation of λ-calculus formalism in ASP as well as talked about ASP representation of FOL-λ-calculus and ASP-λ-calculus. In this section, we will talk about various important λ-calculus operations like substitution, α-conversion, α-equivalence, β-reduction.

We will first begin with our initial approach of implementation of these operations in ASP and will talk about the limitations of such implementation. We will then introduce efficient implementation of these operations using ASP + Lua.

Implementation in ASP

In this section, we will present the implementation of substitution λ-calculus operation and it's limitations.

A variable X in expression E is substituted with the Value Y if there is a need of it. This is encoded in ASP representation as $need(substitution, E, X, Y)$. For eg. if we want to substitute x in $f(loves, x, y)$ by mia, we encode it in ASP as

$$need(substitution, (f(loves, x, y)), x, mia)$$

We want to perform substitution on the sub-expressions too. The following three rules define the need to do substitution on sub-expression of X where X is a function or a λ-expression.

```
need(substitution, Param1, X, Y) :- need(substitution,
                                    f(FunctionName, Param1, Param2), X, Y).
need(substitution, Param2, X, Y) :- need(substitution,
                                    f(FunctionName, Param1, Param2), X, Y).
need(substitution, Function, X, Y) :- need(substitution,
                                      l(Var, Function), X, Y), Var != X.
```

We use sub(E, X, Y, Result) to represent the substitution of X in expression E by Y yields Result. In the following rules, we define the result of substitution in the base case where the expression X is a variable X. We use $not\_var(X)$ to indicate that X is not a variable (X can be a function or a λ-expression).

```
subs(X, X, Value, Value) :- need(substitution, X, X, Value), not not_var(X).
not_var(f(A,B,C)) :- need(substitution, f(A,B,C), Y, Value).
```

```
not_var(l(L,F)) :- need(substitution, l(L,F), Y, Value).
```

The following rule is for a case of replacing variable Y with variable X when X != Y.

```
subs(X, Y, Value, X) :- need(substitution, X, Y, Value),
                        not not_var(X), not not_var(Y), X != Y.
```

If an expression E is a function f(FunctionName, Param1, Param2), then the substitution is performed on E's sub-expressions i.e. Param1 and Param2.

```
subs(f(FunctionName, Param1, Param2), X, Y,
 f(FunctionName, Result1, Result2))
 :-  need(substitution, f(FunctionName, Param1, Param2), X, Y),
         subs(Param1, X, Y, Result1),
         subs(Param2, X, Y, Result2).
```

Similarly, if E is a λ-expression $l(Var, Function)$, then we obtain the substitution result by replacing sub-expression Function of E when substitution variable X is not equal to Var. If variable X is same as Var, E does not change after substitution.

```
subs(l(Var, Function), X, Y, l(Var, ResultFunction))
 :- need(subsitution, l(Var, Function), X, Y),
    Var != X, not not_var(X)
    subs(Function, X, Y, ResultFunction).


subs(l(Var, Function), X, Y, l(Var, Function))
 :- need(substitution, l(Var, Function), X, Y), not not_var(X)
    Var == X.
```

An example, of substitution is $need(substitution, l(y, f(loves, x, y), x, mia)$ i.e. substitute variable x with mia in expression $l(y, f(loves, x, y)$. The result of this substitution will be

$l(y, f(loves, mia, y)$. However, if we want to substitute variable y with mia, then the substitution result will be the same expression $l(y, f(loves, x, y)$ because it is invalid substitution.

Similarly, we can write ASP rules to perform other λ-calculus operations. But there are some serious drawbacks of this implementation. Firstly, we know that, the number of atom in a predicate function in ASP should be fixed. However, our requirement is that we need different number of atoms in a function $f()$. In order to do that, we either need to duplicate ASP code or switch to more complex ASP representation of λ-expressions. Secondly, even for the simple operation like substitution, we needed to define many extra facts such as $need()$ and $not\_var()$.

Implementation in ASP + Lua

In the previous section, we discussed about the implementation of λ-calculus operations in ASP and its serious drawbacks. In order to overcome these drawbacks, we need a tool which would help us solve the problem of predicate function of variable arity and at the same time help us use the features of ASP.

Lua is a lightweight scripting language and it is nicely integrated with ASP. So, with a combination of ASP and Lua we will get an advantage of having the procedural programming (Lua) features as well as Logic programming (ASP) one which exactly suits our need. In this section, we will discuss how efficiently we can use Lua to define the λ-calculus operations and use them easily in ASP like any other predicate symbol in ASP.

Definition of λ-calculus operation in Lua and its usage in ASP is illustrated below.

1. Substitution

   - Lua Function Definition: $substitute(Expression, Variable, Value)$

     where, Expression - λ-expression

     Variable - λ-variable being substituted

     Value - a substitution value

   - Usage in ASP: $result(R) \leftarrow R := @substitute(Expression, Variable, Value).$

     Example: $result(R) \leftarrow R := @subsitute(l(x, f(loves, x, y)), y, mia)$

     The result of above substitution will be $R = l(x, f(loves, x, mia))$.

2. α-conversion

   - Lua Function Definition: $alphaC(Expression, StartIndex[optional])$

     where, Expression - Lambda expression

     StartIndex - start index of generating new λ-variable

   - Usage in ASP: $result(R) \leftarrow R := @alphaC(Expression, StartIndex[optional])$

     Example 1: $result(R) \leftarrow R := @alphaC(l(x, f(woman, x), 2).$

     Here, the resultant expression R will have λ-variable as 'r2'.

     $R = l(r2, f(woman, r2)).$

     Example 2: $result(R) \leftarrow R := @alphaC(l(x, f(woman, x)))$. Here, we did not provide start index. So, the default start index will be 1.

     $R = l(r1, f(woman, r1))$

3. α-equivalence

   - This operation can be easily performed using ASP equality operator. So in order to check the α-equivalence between two expressions, we first use the α-conversion on both the expressions and then check their equivalence.

   - Usage in ASP: $equivalent \leftarrow @alphaC(Expression1) ==$

     $@alphaC(Expression2).$

Example: $equivalent \leftarrow @alphaC(l(x, f(woman, x)) == @alphaC(l(z, f(woman, z)))$.

4. β-reduction

   - Lua Function Definition: $betar(Expression)$

     where, Expression - λ-expression

   - Usage in ASP: $result(R) \leftarrow R := @betar(Expression)$.

     Example: $result(R) \leftarrow R := @betar(a(l(x, x), mia))$

     The result of this β-reduction will be $R = mia$

4.2 Inverse-λ algorithm implementation

In this section, we present the Inverse-λ operators $InverseL$ and $InverseR$ from [2] in ASP. The main aim of these two operators is that given a typed λ-expressions H and G, compute the λ-expression F such that $H = F@G$ or $H = G@F$. We have already discussed about these operators in chapter 1.

Implementation of required operations for Inverse Algorithm

In this section, we will describe the implementation of common set of methods in Lua and implementation of some definitions in ASP which are required for implementation of $InverseL$ and $InverseR$ operators. We have already talked about the λ-calculus operations in section 4.1. All these λ-calculus operations are required for implementation of inverse operators. Apart from these operations, we will need some more set of methods and rules which are illustrated below.

- Replace method in Lua

  – In order to implement the **Definition 1 (operator :)**, we need a method which would help in replacing a term with another one. We define the 'replace' method in Lua which would help us do it.

- Lua Definition: $replace(Expression, Pattern, Value)$

  where, Expression is λ-expression, Pattern is sub λ-expression to be replaced, Value is replacement λ-expression for Pattern.

- The return value of this method is the λ-expression.

- ContainsPattern method in Lua
  - We need this method to check if a term is a part of a λ-expression i.e. if a term is a sub-term of a λ-expression.
  - Lua Definition: $containsPattern(Expression, Pattern)$

    where, Expression is λ-expression, Pattern is a term which we want to check if it belongs to Expression.
  - This method returns $true$ if the Pattern is present in Expression, otherwise returns $false$

- MergeSets method in Lua
  - We have already discussed about Set representation of and, or and $concat$ operators in section 4.1. After performing operations like β-reduction , $replace$, $substitution$ we might have a nested structure of these sets in resultant λ-expression. So we use mergeSets method to merge the nested sets into one set.
  - Lua Definition: $mergeSets(Expression)$

    where, Expression is λ-expression
  - This method returns a λ-expression containing merged sets.

- Sub-terms of a λ-expression
  - A sub-term of a λ-expression L, is any term that occurs in L. In order to work on Inverse operators, we need to know all the possible sub-terms of λ-expression H and G.
  - We define a predicate $subterm(Expression, term)$ in ASP where Expression is a λ-expression and term is a sub-term that occurs in Expression.

– In the below rule we say that any term which is a λ-expression is a sub-term of itself.

```
subterm(L, L) :- term(L).
```

– Further, we say that if any sub-term of L is of the form $l(X, F)$ i.e. a λ-expression containing λ-variable X, then the sub-term of L is F.

```
subterm(L, F) :- subterm(L,l(X,F)).
```

– If a sub-term of L is a λ-application i.e. $a(A1, A2)$ or $A1@A2$, then both $A1$ and $A2$ are the sub-terms of L.

```
2 { subterm(L, A1), subterm(L, A2) } 2 :- subterm(L, a(A1,A2)).
```

– If sub-term of L is of the form $f(C, A1, A2)$ where C is a connector like $imply$ which connects two terms, then both the terms $A1$ and $A2$ are the sub-terms of L.

```
2 { subterm(L, A1), subterm(L,A2) } 2 :- subterm(L, f(C, A1, A2)),
                                         connector(C).
```

– If a sub-term of L is a function symbol of arity n, then all the n arguments of this function are sub-terms of L. We know that arity of a function symbol is not fixed. So, it would be difficult to write a rule in ASP to get the arguments of a function symbol. Hence, we use Lua method $functionSubterm$ to get the arguments of a function symbol. One more difficulty arises, when we return these arguments from $functionSubterm$ method. We cannot return a list of arguments because we would then need to use those many ASP variables to capture these arguments which boils down to the same problem. Hence, we return the arguments of function symbol as a $indent(X, Y)$ predicate function where, X is the argument and Y is a nested $indent(X1, Y1)$ predicate function. Now we can easily get the arguments by writing a recursive rule in ASP. For eg. for the expression $f(h, a, b, c)$, $functionSubterm$ method will return $indent(a, indent(b, indent(c, stop)))$. Also, if the $f()$ is a set of and, or or $concat$ operators then this method returns all possible combinations of the set items.

```
possible_subterm(L,R) :- subterm(L, F), R := @functionSubterm(F),
                         R != null.
possible_subterm(L,Y) :- possible_subterm(L, indent(X, Y)).
subterm(L,X) :- possible_subterm(L,indent(X,Y)).
```

Implementation of InverseL Algorithm in ASP

In the previous section, we have described common set of operations required for $Inverse$ operators. In this section, we will discuss about the implementation of each clause of $InverseL$ definition in ASP.

An input to the $InverseL$ algorithm is defined as $input(inverseL, H, G)$ and the result is $inverseL\_result(F)$.

Clause 1: *If G is $\lambda v.v$, then F = $\lambda v.(v@H)$*

ASP rule for this clause is given below which states that if G is α-equivalent to $\lambda x.x$ then the result is $\lambda x.(x@H)$

```
inverseL_result(l(x, a(x, H))) :- input(inverseL, H, G),
                                  @alphaC(G) == @alphaC(l(x,x)).
```

Clause 2 : *If G is a sub-term of H, then F = $\lambda v.H(G : v)$*

ASP rule for this clause is given below which states that if we replace Pattern G in H1 (α-conversion of H) with variable 'x' and if H1 != R then the result of the inverseL operation is λx.R

```
inverseL_result(l(x, R)) :- input(inverseL, H,G), H1 := @alphaC(H),
                            R := @replace(H1,G, x), H1 != R.
```

Clause 3: *If 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 and 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 $\leq$ kp $\leq$ s, then F = $\lambda w.H(J^1 : w@J^1_{k_1}..@J^1_{k_m}), ..., J^n : w@J^n_{k_1}..@J^n_{k_m})$*

In order to implement this clause, we have multiple set of ASP rules as illustrated below.

```
possible_pattern(H,G, P, a(ID, S), ID) :- input(inverseL,H,G), subterm(H,S),
                                          @alphaC(G) != @alphaC(l(x,x)),
                                          P := @betar(a(G, S)), P != a(G,S),
                                          ID := @generateNewID().
```

Using this rule, we get all the possible patterns/terms generated by applying sub-terms of H to G. Argument P in $possible\_pattern$ is the pattern generated, $a(ID, S)$ is the replacement of P and ID is the λ-variable. For eg. if H is $l(x, f(and, f(woman, mia), f(happy, mia), f(man, vincent), f(happy, vincent)))$ and G is $l(u, l(v, f(and, u, f(happy, v))))$, then one possible pattern P is $l(u, f(and, f(woman, mia), f(happy, u)))$ and its replacement is $w@f(woman, mia)$.

There would be multiple nested λ-applications depending on the number of λ-variables in G. Hence in order to get complete $w@J^1_{k_1}..@J^1_{k_m}$ replacement, we need a recursive rule mentioned below.

```
possible_pattern(H,G,F1, a(P,S),ID) :- possible_pattern(H,G,l(X, F), P, ID),
                                       subterm(H,S),input(inverseL,H,G),
                                       F1 := @mergeSets(@betar(a(l(X,F),S))),
                                       F1 != a(l(X,F), S).
```

Once, we have performed all the λ-applications on expression G to get a complete target expression patterns without λ-variables, then we can capture only valid patterns using below mentioned rule where valid patterns are the terms which are sub-terms of H.

```
valid_pattern(H,G,P,REPLACEMENT,ID) :- possible_pattern(H,G,P,REPLACEMENT,ID),
                                       subterm(H,P).
```

We know that the resultant F expression will have valid pattern P replaced by REPLACEMENT in H. Thus, the possible result is of the form $l(ID, R)$ where ID is the λ-variable and R is the expression formed by replacing P with REPLACEMENT.

```
possible_result(H,G, l(ID, R),REPLACEMENT)
              :- input(inverseL, H,G),
                 valid_pattern(H, G, P, REPLACEMENT,ID),
                 subterm(H,P),R := @replace(H, P, REPLACEMENT), R != H.
```

But, there can be multiple patterns P which needs to be replaced in H. The below rule describes it.

```
2 { possible_result(H,G,R,REPLACEMENT),
    invalid_result(H,G,X,PreviousReplacement) } 2
            :- possible_result(H,G,X,PreviousReplacement),
               valid_pattern(H, G, P, REPLACEMENT, ID),
               input(inverseL,H,G), subterm(H,P),
               @containsPattern(PreviousReplacement, P) != true,
               R := @replace(X,P,REPLACEMENT), R !=X,
               @containsPattern(R, a(ID, REPLACEMENT)) != true.
```

Thus, the result F of $InverseL$ operation is the possible result R such that $H = R@G$.

```
inverseL_result(R) :- possible_result(H,G,R,Z), not invalid_result(H,G,R,Z),
                      R1 := @mergeSets(@betar(a(R,G))),
                      @compare(@alphaC(H) , @alphaC(R1)) == true,
                      input(inverseL,H,G).
```

In the above rule, $compare$ method (defined in Lua) is used to compare the two expressions. We know that the expressions might have sub-term which is a set of and, or or $concat$.

This $compare$ method will check the set equality too which we cannot check directly in ASP using equality operator.

Clause 4: *If H is $\lambda v1..\lambda vi.J$ and $J^1(J^1_{i+1},..J^1_s)$ is a sub-term of J and G is $\lambda w.J(J^1(J^1_{i+1},..J^1_s) : w@J^1_{k_1}...@J^1_{k_s})$ with $\forall$ p, i+1 $\leq$ kp $\leq$ s, then F = $\lambda w.\lambda v_1..v_s.(w@\lambda v_{i+1}...v_s.(J^1(J^1_{i+1},..J^1_s) : v_{k_1},..v_{k_s}))$*

For the implementation of this clause, the main idea is to find the unmatched sub-terms in H and G which does not match with each other i.e. terms of the form $w@J^1_{k_1}...@J^1_{k_s}$ in G and $J^1(J^1_{i+1},..J^1_s)$ in H respectively. In order to find these terms, we first obtain all the matched sub-terms using below rule.

```
matched_pattern(H,G, S) :- input(TYPE, H,G), subterm(H,S), subterm(G,S).
```

We are interested in finding an unmatched pattern in G which is a sub-term of the form $a(a(..,Y)..),Z)$ i.e. $w@J^1_{k_1}...@J^1_{k_s}$.

```
invalid_pattern(G, a(X,Y)) :- subterm(G, a(X,Y)), subterm(G, a(a(X,Y), L)).
unmatched_patternG(H,G,a(X,Y)) :- not matched_pattern(H,G,a(X,Y)),
                                  input(T, H,G),
                                  not invalid_pattern(G,a(X,Y)),
                                  subterm(G,a(X,Y)).
```

In order to find an unmatched pattern in H, we first extract the main target language expression from a λ-expression of H and G i.e. we need to extract J from $\lambda v_1..v_i.J$.

```
mainExpression(H, H) :- input(inverseL,H,G).
mainExpression(G, G) :- input(inverseL,H,G).
mainExpression(E, Y) :- mainExpression(E, l(X,Y)).
```

Now, unmatched pattern in H will be a pattern S such that if we replace the unmatched pattern in G with this S we will get the main expression of H i.e. J.

```
unmatched_patternH(H,G, S) :- input(inverseL, H, G), subterm(H,S),
                              unmatched_patternG(H,G,P),
                              R := @replace(J1, P, S),
                              mainExpression(H,J), J != l(X,Y),
                              mainExpression(H, l(X,Y)),
                              mainExpression(G, J1), J1 != l(X1,Y1),
                              mainExpression(G, l(X1,Y1)),R == J.
```

Once we have found the patterns in H and G respectively, we will need to create an expression F from them. Below rules will create $\lambda v_{i+1}...v_s.(J^1(J^1_{i+1},..J^1_s) : v_{k_1},..v_{k_s}))$ pattern which is a part of expression F

```
internalExpression(H,G, X, l(ID, R)) :- unmatched_patternG(H,G,a(X,G1)),
                                        unmatched_patternH(H,G,H1),
                                        ID := @generateNewID(),
                                        R := @replace(H1, G1, ID), R != H1.

internalExpression(H,G, X, l(ID, R)) :- internalExpression(H,G, a(X, G1), H1),
                                        ID := @generateNewID(),
                                        R :=@replace(H1,G1,ID), R != H1.
```

Now, we need to find valid sub-expression i.e. a sub-expression which does not have any λ-application in it.

```
invalid_internalExpression(H,G, E) :- internalExpression(H, G, a(X, G1), E).
valid_internalExpression(H,G,E) :- internalExpression(H, G, X, E),
                                   not invalid_internalExpression(H,G,E).
```

Below ASP rule will create the resultant expression F which is of the form $\lambda w.\lambda v_1..v_s.(w@\lambda v_{i+1}...v_s.(J^1(J^1_{i+1},..J^1_s) : v_{k_1},..v_{k_s}))$ and checks if $H = F@G$.

```
inverseL_result(l(ID,H1)):- mainExpression(H,J), J != l(X,Y),
                            mainExpression(H, l(X,Y)),
                            valid_internalExpression(H,G,E),
                            ID := @generateNewID(),
                            H1 := @replace(H, J, a(ID, E)), H1 != H,
                            R1 := @mergeSets(@betar(a(l(ID, H1), G))),
                            @alphaC(H) == @alphaC(R1).
```

Implementation of InverseR Algorithm in ASP

In this section, we will discuss about the implementation of each clause of $InverseR$ definition in ASP.

An input to the $InverseR$ algorithm is defined as $input(inverseR, H, G)$ and the result is $inverseR\_result(F)$.

Clause 1: *If G is $\lambda v.v@J$, then F = InverseL(H,J)*

Here, we check that if the expression is of the form $\lambda v.v@J$ then we create an input for $InverseL$ operation i.e. $input(inverseL, H, J)$. In this case, result of $InverseL$ would be the result of $InverseR$.

```
input(inverseL, H, J) :- input(inverseR, H, l(X, a(X, J))),
      G1 := @alphaC(l(X, a(X, J))), G1 == l(r1, a(r1, J)).
inverseR_result(R) :- inverseL_result(R), input(inverseR, H, l(X, a(X, J))).
```

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

The below ASP rule, finds a sub-term J of the expression H such that if we replace J with a λ-variable, it would be α-equivalent to G. The result in this case would be J.

```
inverseR_result(J) :- input(inverseR, H,G), subterm(H,J),
                      @alphaC(G) == @alphaC( l(e1, @replace(H,J, e1))).
```

Clause 3: *If G is not* $\lambda v.v@J$, $J^1(J^1_1, ...J^1_m), J^2(J^2_1, ..J^2_m), .., J^n(J^n_1, .., J^n_m)$ *are sub-terms of H and G is* $\lambda w.H(J^1 : w@J^1_{k_1}..@J^1_{k_m}), ..., J^n : w@J^n_{k_1}..@J^n_{k_m})$ *with 1* $\leq$ *s* $\leq$ *m and* $\forall$*p, 1* $\leq$ *kp* $\leq$ *m, then F =* $\lambda v_1..v_s.J^i(J^i_1, ..J^i_m : v_{k_1}, .., v_{k_m})$

The above clause is valid if G is not $\lambda v.v@J$. The below rule specifies this validity.

```
invalid_Rdefinition3(H, l(X, a(X, J))) :- input(inverseR,H, l(X, a(X, J))).
```

For the implementation of this clause in ASP, we would use most of the ASP rules from the **Clause 3** of $InverseL$ operation. For this clause, we again use the ASP rule from **Clause 3** to find the matched patterns between H and G. Similarly, we would use the ASP rule to find the unmatched pattern in G from **Clause 3** of $InverseL$.

Unlike one unmatched pattern of H in **Clause 3** of $InverseL$, we might have multiple unmatched patterns of H which maps to unmatched pattern in G. So, we need a set of different rules to get all such patterns. First, we find possible unmatched patterns of H using below rule.

```
possible_unmatched_patternH(H,S) :- input(inverseR, H,G),
                                    subterm(H,S), not matched_pattern(H,G,S).
```

Now, the unmatched pattern of H will be a possible unmatched pattern of H and it is not an invalid term i.e. it is not a sub-term of a possible unmatched pattern of H.

```
contains(J,S, X) :- possible_unmatched_patternH(H,S),
                    possible_unmatched_patternH(H,J),
                    X := @containsPattern(J, S), J !=S.
invalid_term(J) :- contains(J,S,true).

unmatched_patternH(H,G,S) :- not invalid_term(S),
                             possible_unmatched_patternH(H,S),
                             input(inverseR,H,G).
```

We again use the rule for creating the internal expression and getting the valid internal expression from **Clause 3** in $InverseL$ algorithm. Thus, the result of $InverseR$ operation, if H and G fits this clause, is defined by below rule

```
inverseR_result(P) :- valid_internalExpression(H,G,P), input(inverseR,H,G),
                      not invalid_Rdefinition3(H,G) .
```

Clause 4: *If H is $\lambda v1..\lambda vi.J$ and $J^1(J^1_{i+1},..J^1_s)$ is a sub-term of J, G is $\lambda w.\lambda v_1..v_s.(w@\lambda v_{i+1}...v_s.(J^1(J^1_{i+1},..J^1_s) : v_{k_1},..v_{k_s}))$, then F = $\lambda w.J(J^1(J^1_{i+1},..J^1_s) : w@J^1_{k_1}...@J^1_{k_s})$*

In order to implement this clause, we first perform the α-conversion on the input expressions.

```
alphaConverted_input(H1,G1) :- input(inverseR, H, l(X,G)),
                               (H1,C1) := @alphaC(H),
                               (G1,C) := @alphaC(G).
```

Below ASP rules are defined to get the main target language expression from the α-converted λ-expression, the sub-terms of α-converted expression of H and mainExpressionG of the form $\lambda v_{i+1}...v_s.(J^1(J^1_{i+1},..J^1_s) : v_{k_1},...v_{k_s})$.

```
mainExpression(H,H) :- alphaConverted_input(H,G).
mainExpression(G,G) :- alphaConverted_input(H,G).
mainExpressionG(G, S) :- mainExpression(G,a(X, S)),
                         alphaConverted_input(H,G).

subterm(H,H) :- alphaConverted_input(H,G).
```

The main idea behind the implementation of this clause in ASP is to find a valid possible result from G such that by performing multiple λ-applications on mainExpressionG using sub-terms of H, we get a valid term which is a sub-term of H. Below rules define this idea.

```
possible_resultG(G, G1, R, a(e1, S)) :- mainExpressionG(G, l(X,G1)),
                                        subterm(H, S),
                                        R := @substitute(G1, X, S),
                                        alphaConverted_input(H,G).

possible_resultG(G, G1, R1, a(A, S)) :- possible_resultG(G,l(X,G1),R,A),
                                        subterm(H,S),
                                        R1 := @substitute(G1, X, S).

invalid_resultG(G, l(X,G1), R, A) :- possible_resultG(G, l(X,G1), R, A),
                                     alphaConverted_input(H,G).

valid_resultG(G, G1, R,A) :- not invalid_resultG(G, G1,R,A),
                             possible_resultG(G, G1, R,A),
                             subterm(H,S), R == S.
```

Here, argument R of $possible\_resultG$ is the reduced term obtained by substituting λ-variable X in G1 with sub-term S of H. G1 is the sub-term of λ-expression G of the form $\lambda x.G1$, while the last argument of the $possible\_resultG$ is the replacement pattern.

Thus, a resultant expression of $InverseR$ operation would be an expression of the form λx.F where F is obtained by replacing pattern R from $valid\_resultG$ with the replacement A.

```
inverseR_result(l(e1, F)) :- valid_resultG(G1, G,R,A),
                             alphaConverted_input(H,G1),
                             F := @replace(H, R, A).
```

Chapter 5

Evaluation of system on Policy Translation

5.1 Experiment Setup

Experiment Data

To evaluate our system, we have created a corpus of iRODS policies. Policies in English were gathered from various sources in iRODS domain and manually translated into IPDL language. We have a set of 100 such policies and their translations. For 10 fold cross validation, we will divide this data into 10 different files each containing a set of 10 policies and their translations. Each training data file has different types of policies. For. eg one type of policy is a policy for printing. An example of one training data file is given in table 5.1. We provide an initial lexicon file to the system.

Configurations

For the NL2KR-L component we have made following configurations to get the best results. We perform 2 iterations over the complete set of training data to learn lexicon (T=2).

Table 5.1: Training Data File example

Policy	IPDL Representation
Generate report listing all preservation attributes	$command > generate(report) \wedge list(report, preservation\_attributes)$
Do not allow renaming of objects in Data folder	$command > \neg allow(rename(Data\_folder(objects)))$
Do not replace AIP template with DIP template	$command > \neg replace(template(aip), template(dip))$
List records that have a data format with an expired lifetime	$command > list(records) \wedge have(records, have(data\_format, expired\_lifetime))$
Verify the existence of required number of replicas	$validation > existence(required(quantity(replicas)))$
Compare staffing level required by a collection to current staffing	$command > compare(required\_by(staffing\_level, collection), current\_staffing)$
Store template for mapping AIP to DIP	$command > store(template(map(aip, dip)))$
Migrate files in Data folder to new storage	$command > migrate(Data_folder(files), storage(new))$
Protect the integrity of Data folder	$command > protect(integrity(Data\_folder))$
Print record containing number of master copy and required replicas	$command > print(record) \wedge contain(record, quantity(master\_copy) \wedge required(replicas))$

For better performance of this component, we provide some restriction configurations in generalization algorithm. For our domain, most of the lexical entries in lexicon have categories N/N and N. In order to prevent blow up of the lexicon with unnecessary categories; while generalizing complete lexicon, we have put these categories in restriction list $LEX\_GENERALIZATION\_EXCLIST$. However, to limit the initial lexicon and to learn similar meanings of nouns, we do not provide category N in $GENERALIZATION\_D\_EXCLIST$. We have put prepositions which might have similar semantics in formal representation, in the list $GENERALIZATION\_D\_PREPOSITIONLIST$. For eg. on, of, in among others, have similar semantics for our domain. Again, to limit generalization of unnecessary meanings of some words, we have put them in exclude word list i.e. $GENERALIZATION\_D\_EXCWORD\_LIST$. Some of the nouns have estranged meanings generated through generalization process which greatly affects performance of the system. In order to limit the training period, we have put these words in an exclude word list.

We assign a default weight of 0.01 to each lexical entry. To tune the weight of each lexical entry, we performed 10 iterations of overall parameter learning algorithm with learning-rate parameters $\alpha_0 = 0.01$ and $c = 0.001$ [17] in Parameter Estimation phase.

CCG Parsing

For our experiments, we use ASPccgTk CCG parser [14] to provide the syntactic parse trees of the sentences. ASPccgTk CCG parser some times gives multiple syntactic parse trees of the sentences. For the better performance of our system we have limited the total number of syntactic parse tree for a sentence to 1.

Most of the policy statements in English are imperative sentences. Hence, there are various ambiguities in the CCG parse of these sentences because the ASPccgTk parser takes CCG categories of words from $C\&C\ Supertagger$ [15] which is trained on CCGBank [20]. So sometimes, the parser might not return good parse trees which will affect the performance of the system. We have first analyzed the parse tree structures for all the sentences in training data set. To get correct CCG parse tree derivation, we have strictly overridden some of CCG

Table 5.2: Sample training corpus

Sentence	IPDL Representation
Generate audit_trail for all changes to rules	$command > generate(audit\_trail(changes(rules)))$
Transfer ownership to rods	$command > transfer(ownership, rods)$
Generate report listing all preservation_attributes	$command > generate(report(list(preservation\_attributes)))$
Migrate files to new storage	$command > migrate(files, storage(new))$
Protect the integrity of Data_folder	$command > protect(integrity(Data\_folder))$
Generate audit_trail for notifications on problems	$command > generate(audit\_trail(notifications(problems)))$
Create AIP template from SIP template	$command > create(template(aip), template(sip))$
Create rule based-on AIP template	$command > create(rule, template(aip))$
On deletion of files from collection erase metadata	$when > deletion(files(collection)); do > command > erase(metadata)$
Generate report summarizing information of micro_services	$command > generate(report(summary(information(micro\_services))))$

categories of words. For eg. the ASPccgTk parser used the category N/N for verb Generate. We have overridden the category of this word to $(S\backslash NP)/NP$. Using such overrides, we were able to get good parse tree derivations for most of the sentences.

Initial Lexicon

Initial lexicon is a key input for our system. To test efficiency of our system, we want to keep the initial lexicon as minimal as possible. In this section, we will give an example of evolution of initial lexicon to final lexicon during the learning process.

Consider a small training corpus of 10 iRODS policies as shown in table 5.2. The CCG derivations of all the sentences in the corpus are given in tables 5.6, 5.7, 5.8 , 5.9, 5.10, 5.11, 5.12, 5.13, 5.14, 5.15. To demonstrate complete learning, we will keep generalization configuration as minimal as possible. We will only maintain $GENERALIZATION\_D\_PREPOSITIONLIST$ and $LEX\_GENERALIZATION\_EXCLIST$ configurations described in previous section. We will now illustrate how performance of the system is affected by initial lexicon by discussing two cases.

Case 1: An initial lexicon for this case is given in table 5.3. Let us talk about the **iteration 1** of learning process. For a first sentence in the corpus, initial lexicon does not contain all the meanings of words. Only meaning of 'Generate' is present in the lexicon. So, inverse algorithm is not helpful in this cases. However, during the learning process, all the nouns 'audit_trail', 'changes', and 'rules' will be generalized and the lexicon will be updated with new meanings $audit\_trail$, $changes$ and $rules$ respectively for these words.

Table 5.3: Case 1: Sample Initial lexicon

Word	CCG	Semantics
Transfer	$(S\backslash NP)/NP$	$\lambda x.\lambda y.command > transfer(x, y)$
ownership	N	$ownership$
rods	N	$rods$
all	$NP/NP; NP/N; N/N$	$\lambda x.x$
the	$NP/NP; NP/N; N/N$	$\lambda x.x$
Generate	$(S\backslash NP)/NP$	$\lambda x.command > generate(x)$

For second sentence in the corpus, lexicon has meanings of most of the words. Inverse algorithm would be helpful in this case. A missing meaning of word 'to' will be learned from this case. The meaning of 'to' learned is $\lambda y.\lambda x.x@y$ and it is updated in lexicon.

For third sentence in the corpus, lexicon does not contain meanings of words 'report', 'listing' and 'preservation_attributes'. Inverse algorithm will not learn meaning of any of these words. However, generalization algorithm will generalize words 'report' and 'preservation_attributes' to $report$ and $preservation\_attributes$. The meaning of word 'listing' will not be learned using generalization because the lexicon does not contain any entry which has the category same as that of 'listing'.

For fourth sentence in the corpus, lexicon contains only meaning of word 'to'. So again here inverse algorithm will not learn anything. Generalization algorithm will help in learning the meaning of words 'Migrate', 'files', 'storage'. Meaning of word 'new' is still not learned through any of these techniques.

For fifth sentence in the corpus, lexicon contains non of the words except 'the' and 'Data_folder'. So again, inverse algorithm will not learn anything. Generalization technique will help in learning meanings of words 'Protect', 'integrity'. Meaning of word 'of' will not be learned.

For sixth sentence, lexicon does not contain words 'for', 'notifications', 'on' and 'problems'. Generalization technique will learn the meanings of words 'notifications' and 'problems' from other words of category N. Meaning of word 'for' will be generalized from meaning of word 'to' because these words are prepositions and they share the same CCG category as shown in

tables 5.7 and 5.11. We have already mentioned in the previous section about generalization configuration for prepositions.

For seventh sentence, inverse algorithm will not learn anything. Generalization technique will help in learning meanings of words ‘Create’, ‘template’, and ‘from’. Preposition ‘from’ will be generalized from words ‘to’ and ‘for’ as they share the same category.

For eighth sentence, generalization technique will learn meanings of words ‘rule’. Meaning of word ‘based-on’ will not be generalized from words having same category as that of ‘based-on’ because these words’ semantics does not contain any predicate names which can be replaced by word ‘based-on’.

For ninth sentence, again generalization technique will learn meanings of words ‘deletion’, ‘collection’, ‘erase’, ‘metadata’.

For tenth sentence, generalization technique will learn meanings of words ‘information’ ‘micro_services’.

After learning from tenth sentence, the first iteration of the algorithm finishes. The updated lexicon after first iteration is shown in the table 5.4. Even though we learned 29 new meanings of words from sample training corpus, most of these meanings are unnecessary. For eg. a new meaning of word ‘deletion’ is $deletion$. But, by looking at an IPDL representation of a sentence, this meaning will not be a good fit. A correct meaning which we want is $\lambda x.deletion(x)$. So, in the **iteration 2** of the learning process, we will not learn anything and we will get the same final updated lexicon.

Thus, if we use this updated lexicon to for translation on the same sample corpus, we will get only one correct result i.e. for sentence 2. Thus an accuracy of the system is 10%. This proves, that this initial lexicon is not good.

Table 5.4: Case 1: Updated lexicon after Iteration 1

Word	CCG	Semantics
Transfer	$(S\backslash NP)/NP$	$\lambda x.\lambda y.command > transfer(x,y)$
ownership	N	$ownership$
rods	N	$rods$
all	$NP/NP; NP/N; N/N$	$\lambda x.x$
the	$NP/NP; NP/N; N/N$	$\lambda x.x$
Generate	$(S\backslash NP)/NP$	$\lambda x.command > generate(x)$ $\lambda x.\lambda y.command > generate(x,y)$
audit_trail	N	$audit\_trail$
changes	N	$changes$
rules	N	$rules$
to	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$
report	N	$report$
preservation_attributes	N	$preservation\_attributes$
Migrate	$(S\backslash NP)/NP$	$\lambda x.command > migrate(x)$ $\lambda x.\lambda y.command > migrate(x,y)$
Data_folder	N	$data\_folder$
storage	N	$storage$
Protect	$(S\backslash NP)/NP$	$\lambda x.command > protect(x)$ $\lambda x.\lambda y.command > protect(x,y)$
integrity	N	$integrity$
for	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$
notifications	N	$notifications$
problems	N	$problems$
Create	$(S\backslash NP)/NP$	$\lambda x.command > create(x)$ $\lambda x.\lambda y.command > create(x,y)$
template	N	$template$
from	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$
rule	N	$rule$
deletion	N	$deletion$
files	N	$files$
collection	N	$collection$
metadata	N	$metadata$
erase	$(S\backslash NP)/NP$	$\lambda x.command > erase(x)$ $\lambda x.\lambda y.command > erase(x,y)$
information	N	$information$
micro_services	N	$micro\_services$

Now let us add some more meanings in the initial lexicon. We will add following lexical entries in the lexicon

1. deletion, N , $\lambda x.deletion(x)$

2. Generate, $(S\backslash NP)/NP$, $\lambda x.\lambda y.command > generate(x@y)$

Case 2: A new initial lexicon is given in table 5.5. **Iteration 1** of the lexicon learning process using this lexicon would be the same as described earlier. However, in this case more meanings will be added in the lexicon. For eg. for a word 'integrity' two meanings will be added i.e. $integrity$ and $\lambda x.integrity(x)$. The updated lexicon after this iteration is described in table A.1 in the appendix.

Now let us discuss **Iteration 2** in detail. For first sentence, we know semantics of all the words except word 'to' with CCG category $(NP\backslash NP)/NP$. Using inverse algorithm semantics of 'to' will be computed and will be updated in the lexicon. For second sentence, we already know all meanings of words. The algorithm will also try to learn new other meanings of words for the given CCG categories. But, all possible semantics of words in a sentence are already learned. So, no new meanings will be learned from this sentence. In case of third sentence, new semantics of word 'listing' will be computed using inverse algorithm because semantics of remaining words is already known. In fourth sentence, now we know meaning of all the words except 'new'. The semantics of 'new' will be learned through inverse-λ algorithm. Similarly, in case of fifth sentence, semantics of word 'of' will be learned through inverse-λ technique. Again, in sentence six, we know meaning of all the words except 'on'. The semantics of 'on' will be learned through inverse-λ technique. One important point to note here is that, semantics of 'on' could have been learned through generalization technique but during our learning process we first prefer inverse-λ approach and if it does not work then we use generalization. In the sentence seven, we do not know the semantics of 'AIP' and 'SIP'. So, inverse-λ algorithm will not learn anything. However, generalization technique will learn semantics of these words from 'new'. Semantics of word 'based-on' in sentence eight, will be learned through inverse-λ technique as semantics of other words are known. In case of sentence nine, we do not know the meaning of 'On' and 'from'. In this sentence, 'from' has CCG category $(NP\backslash NP)/NP$. Being a preposition, the semantics of this word will be learned from word 'of' which has same CCG category. Semantics of 'On' will not be learned through any of these technique in iteration 2. Lastly, in a sentence ten, we will learn meaning of 'summarizing' through inverse-λ technique.

Thus, in this iteration we have learned meanings of all words in the training corpus except word 'On'. However, if we perform **iteration 3** of lexicon learning, the meaning of 'On' will be learned. The final generated lexicon after complete learning process is shown in table A.2. If we evaluate the training corpus over this final generated lexicon, we will get all the correct predictions resulting in 100% accuracy. It implies that our initial lexicon is good.Thus with the addition of just 2 new entries, the accuracy of the system is increased to a great extent.

It is certainly evident that generalization technique plays a crucial role in learning process and helps in maintaining initial lexicon minimal. However, as we increase the corpus size, generalization technique will have negative impact on performance of the system because it might blow up the lexicon with unnecessary meanings. Thus, we will have exponential number of semantic parse trees which would greatly affect performance of the system. Hence, for the larger corpus, we have limited the usage of generalization technique by setting various configurations described in previous sections because of which we had to increase the size of our initial lexicon. We had to add semantics of some words mostly nouns in the initial lexicon even though they could have been learned through generalization technique. For our 10 fold cross validation experiment, our initial lexicon size was approximately 150 for each setup. The final lexicon generated is approximately of size 630.

Even though the initial lexicon plays an important role in defining an accuracy of the system, performance of the system is also greatly affected by CCG parse tree derivations. The CCG derivations of the training corpus described here are idealist cases. Words in different sentences share same categories which helped in learning more words. However, when we use the external CCG parser, CCG derivations of sentences are not good. Sometimes, words have different categories. For eg. word 'of' might have categories $(NP\backslash NP)/NP$ in one sentence while in other sentence of the same pattern as that of first, it will have $(N\backslash N)/NP$, and a parse tree structure of both of these sentences are same. Hence, in this case, we would fail to learn new meanings. Another case is, two sentences have same CCG categories of words but their derivations are different. Hence, again we will not learn much for this case. The only way to cover these cases is to add more meanings in the initial lexicon.

Evaluation Process

To get good results for evaluation of the system on iRODS domain, we need a good initial lexicon. So, we will first start evaluation with a small datasets. Initially we took a dataset of 25 sentences and their translations. In this dataset, all the simple policy statements of the similar kind. The system was trained on this dataset and evaluation was also performed on this same

Table 5.5: Case 2: New updated Initial lexicon

Word	CCG	Semantics
Transfer	$(S\backslash NP)/NP$	$\lambda x.\lambda y.command > transfer(x, y)$
ownership	N	$ownership$
rods	N	$rods$
all	$NP/NP; NP/N; N/N$	$\lambda x.x$
the	$NP/NP; NP/N; N/N$	$\lambda x.x$
Generate	$(S\backslash NP)/NP$	$\lambda x.command > generate(x)$ $\lambda x.\lambda y.command > generate(x@y)$
audit_trail	N	$\lambda x.\lambda y.audit\_trail(x@y)$
deletion	N	$\lambda x.deletion(x)$

Table 5.6: CCG derivation for 'Generate audit_trail for all changes to rules'

Generate	audit_trail	for	all	changes	to	rules
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	NP/N	N	$(NP\backslash NP)/NP$	N
$S\backslash NP$		$(NP\backslash(S\backslash NP))/NP$	NP		$NP\backslash NP$	
$S\backslash NP$		$(NP\backslash(S\backslash NP))/NP$		NP		
$S\backslash NP$		$(NP\backslash(S\backslash NP))$				
	NP					

Table 5.7: CCG derivation for 'Transfer ownership to rods'

Transfer	ownership	to	rods
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	N
$S\backslash NP$		$NP\backslash(S\backslash NP)$	
	NP		

dataset to check that the system learned good meanings of words. In this process, we modify the initial lexicon and generalization configuration to get the best results. The results of this training is described in next section.

Table 5.8: CCG derivation for 'Generate report listing all preservation_attributes'

Generate	report	listing	all	preservation_attributes
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	NP/N	N
$S\backslash NP$		$(NP\backslash(S\backslash NP))/NP$		NP
		$NP\backslash(S\backslash NP)$		
	NP			

Table 5.9: CCG derivation for ‘Migrate files to new storage’

Migrate	files	to	new	storage
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	N/N	N
			N	
$S\backslash NP$		$(NP\backslash(S\backslash NP))/NP$		
$S\backslash NP$			$(NP\backslash(S\backslash NP))$	
NP				

Table 5.10: CCG derivation for ‘Protect the integrity of Data_folder’

Protect	the	integrity	of	Data_folder
$(S\backslash NP)/NP$	N/N	N	$(NP\backslash NP)/NP$	N
$(S\backslash NP)/NP$	N		$NP\backslash NP$	
$(S\backslash NP)/NP$		NP		
$S\backslash NP$				

Table 5.11: CCG derivation for ‘Generate audit_trail for notifications on problems’

Generate	audit_trail	for	notifications	on	problems
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	N	$(NP\backslash NP)/NP$	N
$(S\backslash NP)$		$(NP\backslash(S\backslash NP))/NP$	N	$NP\backslash NP$	
$(S\backslash NP)$		$(NP\backslash(S\backslash NP))/NP$		NP	
$(S\backslash NP)$		$NP\backslash(S\backslash NP)$			
NP					

After, performing evaluation on dataset of 25 policies, we added another set of 25 policies of similar kind to the previous dataset (Now total size of dataset is 50). Again, evaluation is performed on this dataset. Similarly, we performed evaluation of the system, on dataset of 75 and then dataset of 100.

Table 5.12: CCG derivation for ‘Create AIP template from SIP template’

Create	AIP	template	from	SIP	template
$(S\backslash NP)/NP$	N/N	N	$(NP\backslash(S\backslash NP))/NP$	N/N	N
$(S\backslash NP)/NP$	N		$(NP\backslash(S\backslash NP))/NP$		N
$S\backslash NP$			$NP\backslash(S\backslash NP)$		
NP					

Table 5.13: CCG derivation for ‘Create rule based-on AIP template’

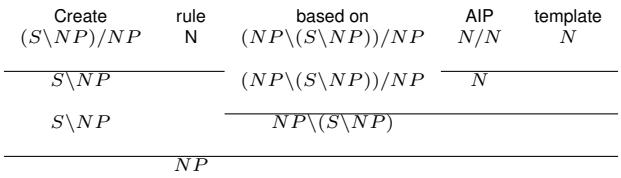


Table 5.14: CCG derivation for ‘On deletion of files from collection erase metadata’

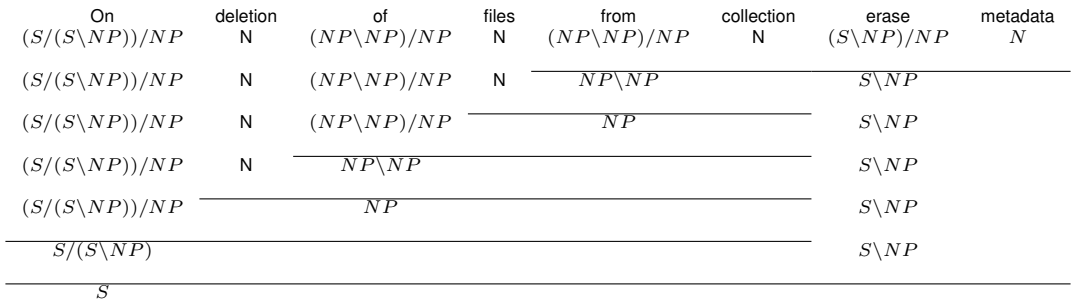
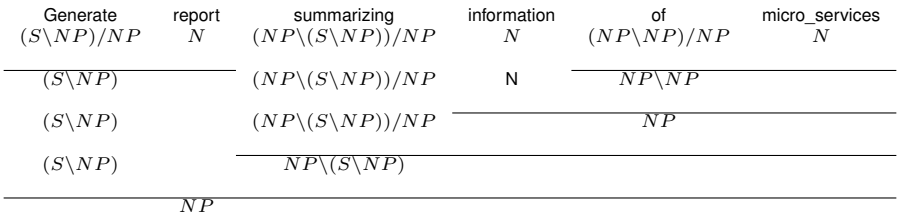


Table 5.15: CCG derivation for ‘Generate report summarizing information of micro_services’



The main aim of training on small datasets and then gradually increase its size is to analyze the initial lexicon and update it with missing meanings which cannot be learned during training process. So, after training the system on complete data set, we will have good initial lexicon.

Once, we have initial lexicon ready, we performed 10 fold cross validation on the dataset of size 100. As already mentioned in the section 5.1, we divided this dataset in 10 chunks of equal size each chunk having unique set of policies. The system was then trained on 9 chunks and evaluated on the remaining 1 chunk, in 10 phases. As a baseline, in order to analyze the performance of the trained system, we evaluated the system using initial lexicon on the same

Table 5.16: Evaluation on training Data

Dataset Size	Precision	Recall
25	95.45%	84%
50	93.33%	84%
75	88.73%	84%
100	90.24%	83.14

Table 5.17: 10-cross fold validation on iRODS domain

System	Precision	Recall
Untrained system (Baseline) (using untrained lexicon)	58.2%	10%
Trained System (using trained lexicon)	68.2%	48%

dataset in 10 fold. While evaluating the system using initial lexicon we maintained a default weight of 0.01 for each lexical entry. The results of this evaluation is described in next section.

5.2 Results and analysis of system performance

We measure the performance of the system using precision and recall where precision is percentage of correct translations returned and recall is percentage of examples with correct translations. The table 5.16 shows the results of evaluation on training dataset of sizes 25, 50, 75 and 100. The results of 10-cross fold evaluation on the system using untrained lexicon and trained one are given in table 5.17.

We can easily see, the trained system performed very well in the evaluation phase. The recall of the untrained system is very low because the initial lexicon didn't have meanings of all words present in a sentence, and the generalization component of the NL2KR-T system did not generate good meanings of such words. However for the trained system, the precision and recall dropped to some extent in 10 fold cross validation as compared to the results obtained during initial lexicon building processing.

System's performance was greatly affected by CCG parser because for some sentences, the parser did not return good CCG derivations. The precision is dropped mainly due to incorrect parse tree derivation and hence the composition of words resulted in incorrect formal

representation. Although, overriding CCG categories for some words improved the CCG parse derivations but not for all sentences. Sometimes, two different sentences of same pattern were parsed differently. For eg. The parser returned different parse tree derivations for the sentences 'Generate audit for changes in micro-services' and 'Print report for changes in micro-services', even though they are of same pattern (same CCG categories). The CCG derivations of these sentences are shown in table 5.18 and 5.19. So, the semantics of some words learned from first sentence might not be a good fit for second sentence because the semantics of words are tightly coupled with their corresponding CCG categories.

Table 5.18: CCG derivation for 'Generate audit for changes in micro-services'

Generate	audit	for	changes	in	micro-services
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	N	$(NP\backslash NP)/NP$	N
$S\backslash NP$		$(NP\backslash(S\backslash NP))/NP$	N	$NP\backslash NP$	
$S\backslash NP$		$(NP\backslash(S\backslash NP))/NP$	NP		
$S\backslash NP$		$(NP\backslash(S\backslash NP))$			
NP					

Table 5.19: CCG derivation for 'Print report for changes in micro-services'

Print	report	for	changes	in	micro-services
$(S\backslash NP)/NP$	N	$(NP\backslash(S\backslash NP))/NP$	N	$(NP\backslash NP)/NP$	N
$S\backslash NP$		$NP\backslash(S\backslash NP)$		$NP\backslash NP$	
NP				$NP\backslash NP$	
NP					

The precision of the trained system is also affected due to incorrect selection of semantics of some words in the context of the sentence which unfortunately had high weight in the lexicon. Although, the weights assigned to each lexicon entry helps in deciding the correct translation of a sentence, sometimes these weights might mislead in picking incorrect lexicon entries and hence we will get incorrect translation. For eg. Consider sentence 'Compare AIP template with SIP template'. In this sentence, a word $with$ is used in the context of comparison. The correct semantics of $with$ present in the lexicon in this context is $\lambda y.\lambda x.x@y$. Now consider the sentence 'List all people with access permission on collection'. In this sentence,

a word $with$ is used in the context of characterization or having some property. The semantics of $with$ in the lexicon for this context is $\lambda y.\lambda x.x@have(y)$. The CCG category of word $with$ is same for both of these semantics. As the training data set has more sentences containing word $with$ in the context of comparison, the weight assigned to the lexicon entries for word $with$ was more biased towards the one having semantics $\lambda y.\lambda x.x@y$. Hence, NL2KR-T system translated some sentences containing word $with$ in the context of characterization, incorrectly.

Another, most important factor which affected the performance of the system, typically in terms of time complexity is the number of semantic parse trees for each sentence. We know that a word with a given CCG category can have multiple meanings. Consider a sentence 'List micro-services referenced by a rule'. If each of the words in this sentence has 5 different meanings for their CCG category, the total number of parse trees having unique set of semantics for each node will be $5 \times 5 \times 5 \times 5 \times 5 \times 5$ i.e. 15625. Thus, for one CCG parse tree derivation of a sentence, we can get 15625 different trees. For multiple CCG parse derivations of a sentence, the total number trees will increase in multiple of the number of CCG parse trees. During the learning phase, with every iteration, the number of meanings in the lexicon might increase. Hence, again the number of parse trees will increase by that factor. Processing of those many parse trees takes great amount of time and it is just for a one sentence. Thus, the performance of parameter estimation component in NL2KR-L and PCCG computation component in NL2KR-T is greatly affected. Typically, the computation of feature vector for each parse tree takes huge computation time as well as memory. If we have approximately 600 entries in the lexicon and approximately 15,000 semantic trees for a sentence, then the computation of feature vector for all the parse trees of one sentence will take 15,000*600 iterations. By addition of more lexical entries in lexicon, the number of iterations quickly grow exponentially.

Chapter 6

Conclusion and Future Work

Translation of natural language to a formal representation plays a key role in communication between a computer based system and human beings. Sometimes, an interface langauge for a computer based system is very complicated and direct translation of natural language to this language becomes difficult. An example of one such computer based system is iRODS which uses rule oriented language for interaction purpose. In this thesis, we have designed an Intermediate Policy Declarative Language (IPDL) for this system such that natural language can be translated to formal statements in IPDL and further it can translated to iRODS rules. For the purpose of translation of natural language to a formal representation, we have developed an improved version of NL2KR system (v.2) over NL2KR v.1 system [1] by improving and re-developing various modules of this system. Further, we have evaluated improved version of the system on iRODS domain and analysed performance of the system.

The NL2KR system's key item is a CCG parse tree of a sentence. v.1 of the system uses an inbuilt CCG parser with some implementation drawbacks. In v.2 of the system, we re-developed this inbuilt CCG parser in ASP and also integrated the external CCG parser (AS-PccgTk [14]) with the system and provided a configuration functionality to use either of them. Other components of the system like $Inverse - \lambda$, $Generalization$ were also improved and new features were added to them. In v.1, the learning algorithm of NL2KR-L sub-system, has some major drawbacks. It was learning only missing semantics of words in a sentence. In v.2, we have re-designed this algorithm. Apart from this, we have fixed some major issues in the system and introduced some new features like better system configurations to handle the components, memory cache to improve computation time of the system.

While improving the system, we noticed that ASP would be a good fit for the implementation of some of the modules of the system because it can easily solve a search problem by defining only few rules and, by having these components in ASP, we can easily use knowledge to improve performance of the system. One of the major contribution of this research is ASP

representation of λ-expressions and implementation of λ-operations like α-conversion, β-reduction, substitution in ASP. We have also implemented Inverse-λ component in ASP.

We evaluated NL2KR v.2 system on iRODS domain and analysed it's performance. iRODS rule oriented language has syntactic structure like a procedural language. So we have designed a simple Intermediate Policy Declarative Language (IPDL) for this system. Using NL2KR v.2 system, we first train it with a set of policies in natural language (English) and their corresponding IPDL translations and then we test the system by translating new policies in natural language.

While this work provides various improvements in the NL2KR system, there are several areas which could be desired extensions of this research. In the next sections, we will discuss about few directions in which this research can extend.

6.1 Using world knowledge to enhance system performance

In the previous chapter, we have talked about the problem with number of semantic parse trees. We know that, out of total parse trees of a sentence only few will generate a correct composition of a formal representation. While generating all the possible parse trees, we only look at CCG categories of words to pick their possible semantics. Instead of just using CCG categories to pick semantics, we can use CCG categories and Sense of a word in the context of a sentence, to pick possible semantics. Thus, we can easily limit number of parse trees and would increase the system performance. The idea is to have one more entry in the lexicon for Sense for each lexical item. So while parsing a sentence, we can disambiguate senses of words/phrases in it and pick possible semantics of these words from lexicon using their CCG categories and Senses together.

Another important use of world knowledge would be during translation phase. As discussed in previous section, the NL2KR-T system returned incorrect translation of some sentences due to incorrect selection of semantics of some words in the context of a sentence which unfortunately had high weight in the lexicon. Again, for such cases, knowing senses of semantics of words would help in picking correct translation even though it has less probability.

Table 6.1: Pattern Matching Example

Sentence	Translation
Print staff experience report	$command > print(report(staff\_experience))$
Print financial audit report	$command > print(audit\_report(finance))$
Print preservation metadata template	$command > print(template(preservation\_metadata))$

We know that ASP is good at representing knowledge and is easily extensible to new knowledge. We can use ASP to disambiguate senses of words in a sentence using knowledge. We have already taken a step ahead by representing λ-expressions and implementing their basic operations in ASP in this research. An extension of this research could be creating a generic framework to use the world knowledge to guess particular semantics of a word in the context of a sentence.

6.2 Improvements in NL2KR system

Our NL2KR-L learning component of the system, uses $Inverse - \lambda$ and $Generalization$ techniques to learn new meanings of words. However, both these techniques are not enough to learn new meanings of words. $Inverse - \lambda$ technique will learn a meaning of a word in a sentence provided meanings of other words in the sentence are present. $Generalization$ technique may produce large number of unnecessary semantics which might not help in composing meaning of a sentence. Sometimes, both of these techniques do not help in learning new meanings of a word. Typically this happens when both the children of a node does not have semantics and non of them are leaf nodes. Thus, these techniques do not help much to keep our initial lexicon minimal.

In research [18], a pattern matching approach is described. We can use similar approach that uses parse trees of sentences and their corresponding translations in training corpus, and look for patterns in them and learn lexicon. For eg. Consider some examples of iRODS policies in table 6.1. By analyzing patterns in these examples, we can learn that meaning of print could be $\lambda x.command > print(x)$.

Another semantics learning technique we can integrate with our system is Higher Order Unification. Although Higher Order Unification is undecidable, a restricted version of Higher

Order Unification is described in [19]. The idea is to find the λ-expressions of the children given a λ-expression of parent such that when applied produces same λ-expression of parent.

REFERENCES

[1] Baral, C.; Dzifcak, J.; Gonzalez, M.; and Zhou, J. 2011. Using Inverse Lambda and Generalization to Translate English to Formal Languages. In *Proceedings of International Conference on Computational Semantics (IWCS) 2011, Oxford*

[2] Gonzalez, M. 2010. An inverse lambda calculus algorithm for natural language processing. *Master's thesis, Arizona State University*.

[3] Baral, C. *Knowledge Representation, Reasoning, and Declarative Problem Solving*. Cambridge University Press (2003).

[4] Zettlemoyer L.;Collins M. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars. In *Proc. of UAI-2005, Edinburgh,Scotland, July*

[5] Lobo J.;Bhatia R.;Naqvi S. A policy description language. *In Proc. of AAAI, Orlando, FL, July 1999*

[6] Gelfond M.; Lifschitz, Representing action and change by logic programs. *In the Journal of Logic Programming, Volume 17,November 1993*

[7] Niemelä I.;Simons P. Smodels-an implementation of the stable model and a well-founded semantics for normal logic programs. *In: LPNMR, 1997, 420-429*

[8] Leone N.; Pfeifer G.; Faber W.; Eiter T.; Gottlob G.; Perri S.; Scarcello F. The DLV system for knowledge representation and reasoning. *In ACM Transactions on Computational Logic 7(3) (2006) 499-562*

[9] Gebser M.; Kaminski R.; Kaufmann B.; Ostrowski M.; Schaub T.; Thiele S. A User's Guide to gringo, clasp, clingo, and iclingo. (2008)

[10] Montague R. *Formal Philosophy. Selected Papers of Richard Montague*. Yale University Press, 1974

[11] Clark S.; Curran J. Log-linear models for wide-coverage CCG parsing. *In: SIGDAT Conference on Empirical Methods in Natural Language Processing, EMNLP, 2003*

[12] Clark S.; Curran J. Parsing the WSJ using CCG and log-linear models. *In: Proceedings of the 42nd Annual Meeting of the Association for Computational Linguistics (ACL 2004), Barcelona, Spain*

[13] Rajasekar A.;Moore R.;Hou C.;Lee C.;Marciano R.;Torcy A.;Wan M.;Schroeder W.;Chen S.; Gilbert L. iRODS primer: integrated rule-oriented data system. *Synthesis Lectures on Information Concepts, Retrieval, and Services 2, 2010*

[14] Erdem E.; Lee J. ;Lierler Y.; Pearce D. Parsing Combinatory Categorial Grammar via Planning in Answer Set Programming *Correct Reasoning, Lecture Notes in Computer Science Volume 7265, 2012*

[15] Stephen C.; James C. The Importance of Supertagging for Wide-Coverage CCG Parsing. *Proceedings of the 20th International Conference on Computational Linguistics (COLING-04), Geneva, Switzerland, 2004*

[16] Church A. âĂIJA formulation of the simple theory of types,âĂİ *The Journal of Symbolic Logic, vol. 5, no. 2, pp. 56âĂŞ68, 1940*

[17] Zettlemoyer L.; Collins M. Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars *In Proceedings of the Twenty-First Conference on Uncertainty in Artificial Intelligence, 2005*

[18] Kwiatkowski T.; Zettlemoyer L.; Goldwater S.; Steedman M. Inducing probabilistic ccg grammars from logical form with higher-order unification *In Proceeding of the Conference on Empirical Methods in Natural Language Processing pp 1223-1233, 2010*

[19] Kwiatkowski T.; Zettlemoyer L.; Goldwater S.; Steedman M. Lexical generalization in ccg grammar induction for semantic parsing *In Proceeding of the Conference on Empirical Methods in Natural Language Processing pp 1512-1523, 2011*

[20] Hockenmaier J.; Steedman M. Acquiring compact lexicalized grammarsfrom a cleaner treebank. *In Proceedings of the Third LREC Conference, pages 1974âĂŞ1981, Las Palmas, Spain,2002*

Appendix A

Case 2: Updated and Final Lexicons

Table A.1: Case 2: Updated new lexicon after Iteration 1

Word	CCG	Semantics
Transfer	$(S\backslash NP)/NP$	$\lambda x.\lambda y.command > transfer(x,y)$
ownership	N	$ownership$
rods	N	$rods$
all	$NP/NP; NP/N; N/N$	$\lambda x.x$
the	$NP/NP; NP/N; N/N$	$\lambda x.x$
Generate	$(S\backslash NP)/NP$	$\lambda x.command > generate(x)$ $\lambda x.\lambda y.command > generate(x,y)$ $\lambda x.\lambda y.command > generate(x@y)$
audit_trail	N	$audit\_trail$ $\lambda x.audit\_trail(x)$
changes	N	$changes$ $\lambda x.changes(x)$
rules	N	$rules$ $\lambda x.rules(x)$
to	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$
report	N	$report$ $\lambda x.report(x)$
preservation_attributes	N	$preservation\_attributes$ $\lambda x.preservation\_attributes(x)$
Migrate	$(S\backslash NP)/NP$	$\lambda x.command > migrate(x)$ $\lambda x.\lambda y.command > migrate(x,y)$
Data_folder	N	$data\_folder$ $\lambda x.data\_folder(x)$
storage	N	$storage$ $\lambda x.storage(x)$
Protect	$(S\backslash NP)/NP$	$\lambda x.command > protect(x)$ $\lambda x.\lambda y.command > protect(x,y)$
integrity	N	$integrity$ $\lambda x.integrity(x)$
for	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$
notifications	N	$notifications$ $\lambda x.notifications(x)$
problems	N	$problems$ $\lambda x.problems(x)$
Create	$(S\backslash NP)/NP$	$\lambda x.command > create(x)$ $\lambda x.\lambda y.command > create(x,y)$
template	N	$template$ $\lambda x.template(x)$
from	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$
rule	N	$rule$ $\lambda x.rule(x)$
deletion	N	$deletion$ $\lambda x.deletion(x)$
files	N	$files$ $\lambda x.files(x)$
collection	N	$collection$ $\lambda x.collection(x)$
metadata	N	$metadata$ $\lambda x.metadata(x)$
erase	$(S\backslash NP)/NP$	$\lambda x.command > erase(x)$ $\lambda x.\lambda y.command > erase(x,y)$ $\lambda x.\lambda y.command > erase(x@y)$
information	N	$information$ $\lambda x.information(x)$
micro_services	N	$micro\_services$ $\lambda x.micro\_services(x)$

Table A.2: Case 2: Final lexicon after Iteration 3

Word	CCG	Semantics	Weight
Transfer	$(S\backslash NP)/NP$	$\lambda x.\lambda y.command > transfer(x,y)$	0.07646726
		$\lambda x.\lambda y.command > transfer(x@y)$	-0.024746018
		$\lambda x.command > transfer(x)$	-0.024746014
ownership	N	$ownership$	0.07570592
		$\lambda x.ownership(x)$	-0.023981703
rods	N	$rods$	0.07493635
		$\lambda x.rods(x)$	-0.023250459
all	$NP/NP; NP/N; N/N$	$\lambda x.x$	0.107222944
		$\lambda x.x@all$	-0.08715378
the	$NP/NP; NP/N; N/N$	$\lambda x.x$	0.059751213
		$\lambda x.x@all$	-0.03880422
Generate	$(S\backslash NP)/NP$	$\lambda x.command > generate(x)$	-0.11762427
		$\lambda x.\lambda y.command > generate(x,y)$	-0.11698096
		$\lambda x.\lambda y.command > generate(x@y)$	0.26020452
audit_trail	N	$audit\_trail$	-0.088598415
		$\lambda x.audit\_trail(x)$	0.10562369
changes	N	$changes$	-0.041762702
		$\lambda x.changes(x)$	0.0580377
rules	N	$rules$	0.05806967
		$\lambda x.rules(x)$	-0.041730717
to	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$	0.10719467
	$(NP\backslash NP)/NP$	$\lambda y.\lambda x.x@y$	-0.0895291
report	N	$report$	-0.08859752
		$\lambda x.report(x)$	0.105146274
preservation_attributes	N	$preservation\_attributes$	0.05841183
		$\lambda x.preservation\_attributes(x)$	-0.041189767
Migrate	$(S\backslash NP)/NP$	$\lambda x.command > migrate(x)$	-0.02325855
		$\lambda x.\lambda y.command > migrate(x,y)$	0.076641545
		$\lambda x.\lambda y.command > migrate(x@y)$	-0.02325855
Data_folder	N	$data\_folder$	0.05847109
		$\lambda x.data\_folder(x)$	-0.041031405
storage	N	$storage$	-0.03994585
		$\lambda x.storage(x)$	0.05995425
Protect	$(S\backslash NP)/NP$	$\lambda x.command > protect(x)$	0.07548905
		$\lambda x.\lambda y.command > protect(x,y)$	-0.024013432
		$\lambda x.\lambda y.command > protect(x@y)$	-0.024013432
integrity	N	$integrity$	-0.041033715
		$\lambda x.integrity(x)$	0.058468774
for	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$	0.006702896
notifications	N	$notifications$	-0.04007826
		$\lambda x.notifications(x)$	0.059325315
problems	N	$problems$	0.059282325
		$\lambda x.problems(x)$	-0.04012125
Create	$(S\backslash NP)/NP$	$\lambda x.command > create(x)$	-0.055058286
		$\lambda x.\lambda y.command > create(x,y)$	0.14011657
		$\lambda x.\lambda y.command > create(x@y)$	-0.055058286
template	N	$template$	-0.13425219
		$\lambda x.template(x)$	0.1537129
from	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$	0.07550978
from	$(NP\backslash NP)/NP$	$\lambda y.\lambda x.x@y$	-0.057941414
rule	N	$rule$	0.059181765
		$\lambda x.rule(x)$	-0.040024586
deletion	N	$deletion$	-0.040011086
		$\lambda x.deletion(x)$	0.060000047
files	N	$files$	0.01237214
		$\lambda x.files(x)$	0.00748118
collection	N	$collection$	0.05986771
		$\lambda x.collection(x)$	-0.040132284
metadata	N	$metadata$	0.05870446
		$\lambda x.metadata(x)$	-0.041295536
erase	$(S\backslash NP)/NP$	$\lambda x.command > erase(x)$	0.075812176
		$\lambda x.\lambda y.command > erase(x,y)$	-0.024187827
		$\lambda x.\lambda y.command > erase(x@y)$	-0.024187835
information	N	$information$	-0.040053308
		$\lambda x.information(x)$	0.059054725
micro_services	N	$micro\_services$	0.05901827
		$\lambda x.micro\_services(x)$	-0.040089756
listing	$(NP\backslash(S\backslash NP))/NP$	$\lambda z.\lambda x.x@\lambda y.list(z@y)$	0.009448431
new	N/N	$\lambda x.x@new$	0.00976438
of	$(NP\backslash NP)/NP$	$\lambda y.\lambda x.x@y$	0.0059644114
on	$(NP\backslash NP)/NP$	$\lambda y.\lambda x.x@y$	0.009079316
based-on	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@y$	0.009168728

On	$(S/(S\backslash NP))/NP$	$\lambda x.\lambda y.when x; do > y$	0.009695122
summarizing	$(NP\backslash(S\backslash NP))/NP$	$\lambda y.\lambda x.x@summary(y)$	0.008973512

Appendix B

ASPccgTK Parser details

- **Parser configuration in config.properties**

  1. $CCGPARSER = ASPccgTk$: Set this configuration to use ASPccgTK parser
  2. $ASPccgTk\_Strict\_Override = true$: If you want to use strict override of CCG categories in the parser, set $ASPccgTk\_Strict\_Override$ to true otherwise false to use suggestion override
  3. $ASPccgTk\_SUPERTAGGER = supertag.py$: If you want to use the c & c parser parts of speech tagger, set the configuration to 'supertag.py' and if you want to use the Stanford Parts-of-speech tagger, set the configuration to 'stanfordsupertag.py'

- nl2kr_parsetree.asp is the wrapper ASP program over ASPccgTk parser.

- The supertagged files are saved in examples_supertagged folder in resources/aspccgtk-parser directory.

- The output of the parsing is saved in output folder in resources/aspccgtk-parser directory.

- To run the parser from command prompt, use following command

  python supertag.py -s='<Sentence>' -syntax_override=<syntax file path>
  -strict_override=<True/False>

  python parse_and_display.py examples_supertagged/<supertagged sentence file>
  -nl2krSemDict=<semantics dictionary path>

Appendix C

How to run the NL2KR v.2 system

1. **Training System (NL2KR-L)**

   - To train the NL2KR system run the 'bioai.ccgprocessor.training.TrainingSystem.java' file with the following arguments

     -train -data=<training data file1>,<taining data file 2>,.. -dict=<initial lexicon1>,<initial lexicon1>,... -syntax=<syntaxfile1>,<syntaxfile2>... -output=<final lexicon location>

   - Training data file: training data file contains sentences and their formal representation seperated by TAB
     eg.
     Vincent loves Mia loves(vincent, mia)
     Some boxer walks EX. (boxer(X) :- walks(X))

   - Syntax dictionary: Syntax dictionary contains CCG categories of words. Each entry contains word followed by its CCG. Both are seperated by TAB
     eg.
     Vincent NP
     Mia NP
     takes $(S\backslash NP)/NP$

   - Initial lexicon: Initial lexicon contains semantics of words. Each entry contains word, its CCG and its semantics in terms of lambda expression. All are seperated by TAB
     eg.
     Vincent $NP\ vincent$
     Mia $NP\ mia$
     takes $(S\backslash NP)/NP\ \#w.\#z.(w@x.takes(z,x))$

   - OUTPUT: The output of the training the system is final lexicon file 'dictionary_train.out' in output folder.
   - NOTE: Training system uses 'clingo' for parse tree generation. Make sure that your clingo is executable. You can find executable clingo in 'resources' directory

2. **Translation System (NL2KR-T)**

   - To test the NL2KR system run the 'bioai.ccgprocessor.testing.TestingSystem.java' file with the following arguments

     -train -data=<test data file1>,<test data file 2>,.. -dict=<final lexicon1>,<final lexicon1>,... -syntax=<syntaxfile1>,<syntaxfile2>...

- Test data: Test data file contains the sentences and their formal representation separated by TAB. It is similar to training data file
- Final Lexicon: Final lexicon is the lexicon generated by the training system.

3. **Example**

- Sample example files are given to show how the system works. Sample files are
  sample_dict.txt
  sample_syntax.txt
  sample_train.txt
  sample_test.txt

- For training : Run TrainingSystem.java file with arguments
  -train -data=sample_train.txt -dict=sample_dict.txt -syntax=sample_syntax.txt
  OR
  -train -data=corpora/BioKR/bio_train.txt -syntax=corpora/BioKR/bio_syntax.txt
  -dict=corpora/BioKR/bio_dict.txt -output=corpora/BioKR/bio_trained.txt
- For testing: Run TestingSystem.java file with arguments
  -test -data=sample_test.txt -dict=dictionary_train.out -syntax=sample_syntax.txt
  OR
  -test -data=corpora/BioKR/bio_test.txt -dict=corpora/BioKR/bio_trained.txt
  -syntax=corpora/BioKR/bio_syntax.txt