Answering Deep Queries Specified in Natural Language with Respect to a Frame Based Knowledge Base and Developing Related Natural Language Understanding Components

by

Nguyen Ha Vo

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved August 2015 by the
Graduate Supervisory Committee:

Chitta Baral, Chair
Joohyung Lee
Kurt VanLehn
Tran Cao Son

ARIZONA STATE UNIVERSITY

December 2015

ABSTRACT

Question Answering has been under active research for decades, but it has recently taken the spotlight following IBM Watson's success in *Jeopardy!* and digital assistants such as Apple's Siri, Google Now, and Microsoft Cortana through every smart-phone and browser. However, most of the research in Question Answering aims at factual questions rather than deep ones such as "How" and "Why" questions.

In this dissertation, I suggest a different approach in tackling this problem. We believe that the answers of deep questions need to be formally defined before found. Because these answers must be defined based on something, it is better to be more structural in natural language text; I define Knowledge Description Graphs (KDGs), a graphical structure containing information about events, entities, and classes. We then propose formulations and algorithms to construct KDGs from a frame-based knowledge base, define the answers of various "How" and "Why" questions with respect to KDGs, and suggest how to obtain the answers from KDGs using Answer Set Programming. Moreover, I discuss how to derive missing information in constructing KDGs when the knowledge base is under-specified and how to answer many factual question types with respect to the knowledge base.

After having the answers of various questions with respect to a knowledge base, I extend our research to use natural language text in specifying deep questions and knowledge base, generate natural language text from those specification. Toward these goals, I developed NL2KR, a system which helps in translating natural language to formal language. I show NL2KR's use in translating "How" and "Why" questions, and generating simple natural language sentences from natural language KDG specification. Finally, I discuss applications of the components I developed in Natural Language Understanding.

DEDICATION

To Mom and Dad, Luong Vo and Trang Ha; it's impossible to thank you adequately for everything you've done, from loving me unconditionally to encouraging me to fulfill our dreams.

To my wife, Bich Bui and my daughters, Annie and Serene Vo, for their unwavering understanding and endless support.

ACKNOWLEDGMENTS

I would like to express my deepest appreciation for my mentor Dr. Chitta Baral, his invaluable guidance, and continuous encouragement.

Dr. Yonggwan Won, who has been an advisor and a second father to me since I began my masters program; it is through him that I began learning about research - thank you.

Thank you to my committee members, Dr. Joohyung Lee, Dr. Tran Cao Son, and Dr. Kurt VanLehn, for their valuable comments and feedback.

I would like to take this opportunity to express gratitude for Arpit Sharma, Arindam Mitra, and all of my other colleagues in BioAI lab for their help and support.

TABLE OF CONTENTS

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

CHAPTER Page

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 Motivation

Question Answering (QA) has been an active research area for decades (Maybury, 2004; Strzalkowski and Harabagiu, 2006; Mendes and Coheur, 2012; Gupta *et al.*, 2012), but recent success of Question Answering systems, such as IBM Watson (Ferrucci *et al.*, 2010), Apple's Siri (Aron, 2011), Google Now (Google, 2012) and Microsoft Cortana, has been bringing the concept closer to the public than ever.

Until now, most of the work in QA concentrate on factual questions (Gupta *et al.*, 2012). These questions are usually answered by looking for explicitly stated facts in text or/and knowledge bases. The approaches along this line have been widely investigated and archived; many notable successes such as Watson, which defeated *Jeopardy!*'s former winners and received first prize of $1 million in 2011 or the AURA System (Chaudhri *et al.*, 2009) in project Halo, which passed the AP Biology test in 2004.

However, there are very few works on deep question answering; works that exist have a limited view (Higashinaka and Isozaki, 2008; Aouladomar, 2005; Verberne, 2009, 2006). These works attempted to answer "Why" questions by **searching** for the answers in the given corpus, based on finding terms such as "because," "so," in sentences like " A because B" or "Because of B, A". These approaches fail when the corpus does not contain such terms or the answer needs to be **constructed** from many parts within the corpus. Moreover, these approaches do not work when the indication terms are not sufficient.

1.2 Specific Research Contributions

As current approaches of searching for answer of deep questions have aforementioned drawbacks, we think a more suitable approach should "construct" the answers rather than simply search. The major steps in developing our system answering deep questions (P1 to P5 in Figure 1.1) are as follow:

Formulation:

- P1: Consider the text about photosynthesis in the top left of figure 1.1 and the deep question, "How does the Calvin cycle work?". This question can not be answered by searching the text using indication terms such as "because," "so," and "by" in straightforward search patterns like "The Calvin cycle works by ...." However, the answer (such as the one in the bottom left of figure 1.1) may be obtained by combining many facts scattered throughout the text. But, to know how to combine such facts, one first needs to formally define the answer to such a "How" question. Formally defining this is the first step of the thesis, and we refer to it as P1.

- P2: Because such answers must be defined based on some knowledge, in a particular knowledge representation, we explored various knowledge representation schemes. This knowledge representation should be more structural than a natural language and should support inference, an imperative property for answering many deep questions. Developing such a representation is the second step in the thesis, and we refer to it as P2. In addressing it, we formulated the notion of Knowledge Description Graphs (KDGs). This structure contains information about events, entities, classes, and the relations among them. In the middle right of figure 1.1, we give an example KDG corresponding to the text in the top left. In the bottom right is an answer to the question, "How does the Calvin cycle work?" constructed with respect to the example KDG.

Major tasks:

- P3: The third step of the thesis is to construct KDGs from knowledge bases, which we name P3. Since constructing KDGs directly from natural language text is extremely challenging, we begin with frame-based knowledge bases such as AURA. Additionally, this step includes enriching the knowledge bases if needed. We consider two cases of missing information as follows:

  1. In object-oriented frame-based knowledge bases, the information of each instance is compactly represented using inheritance. To get the complete information, one must utilize a process called "unification."
  2. In many cases, the knowledge base is missing information, such as type of instances or order of events. However, such information can be derived from related facts.

  For each case, we suggest formulations and show an implementation in Answer Set Programming for enrichment. We also show how to answer many factual question types with respect to the knowledge base (AURA).

- P4: Using the KDGs constructed from knowledge bases and the answers formally defined with respect to KDGs, our next step is to construct the answers of deep questions from KDGs. We refer to it as P4, which we address by showing an implementation in Answer Set Programming and by proving its correctness.

- P5: So far, we have the answers of various factual and deep question types with respect to a knowledge base. Our next goal is to extend our research to use natural language text. While our first target is to only translate deep questions in natural language to some formal representation, our long-term target is much more ambitious and challenging: translating natural language text to KDGs. This long-term target is

our next step, referred to as P5. We achieved the first target using NL2KR, the system we developed to aid in translating natural language to formal language. However, P5 requires much more work in upcoming years to have substantial outcomes. For example, Knowledge Parser, which is not included in this thesis but will be discussed later in the conclusion, is our first attempt towards it. Knowledge Parser combines NL2KR and many other techniques to translate natural language text into KDGs.

After solving the pieces above, we realize that our approach can be used to tackle the bigger related problem: Natural Language Understanding.

In the following chapters, we describe modules to address the puzzle pieces P1-P5. After that, in the last chapter, we discuss applications and the path to solve some Natural Language Understanding problems. Details about the content of each chapter and the progress of the thesis is presented in the following section.

1.3 Organization of the Dissertation

Following is the list of chapters in the dissertation:

- Chapter 1 introduces the motivation and modules of this thesis.
- Chapter 2 defines the KDG and its use in answering deep questions. It addresses P1, P2, and part of P4
- Chapter 3 solves the problem of cloning and unification, which addresses most of P3. Although we employ the same preprocessing module as (Baral and Liang, 2012) to convert text in skolemized KM format (Clark *et al.*, 2004) to Answer Set Programming, we utilize different formulations and algorithms to solve the cloning and unification problem. We provide details about the implementation and prove various formulations. Additionally, we improve most of the factual question answering in

(Baral and Liang, 2012); for the rest, we implement similar answers using our new formulations.

- Chapter 4 shows the implementations of P2 and of deep question answering using Answer Set Programming, addressing another part of P4.

- Chapter 5 discusses about reasoning to fill the missing information in many cases where the knowledge base used to construct KDGs is incomplete.

- Chapter 6 introduces the NL2KR system, which can be used to build systems that translate natural language to formal language. This module aims to address P5. We brought NL2KR from a proof-of-concept state in (Kumbhare, 2013) to the prototype state by including various new features and improvements such as installation packages, GUI, and bug fixing. Our other major contributions in NL2KR are (i) CCG parser, which is based on planning, and (ii) new iterative semantic learning algorithms. The improvements not only make NL2KR usable for people who have no previous experience on the subject matter but also greatly speed up the learning process. For example, learning on the corpus of 100 sentences now takes about 3 minutes while previous versions took more than 10 hours without finishing.

- Chapter 7 concludes the contribution in this dissertation and discuss the applications of the techniques presented in previous chapters.

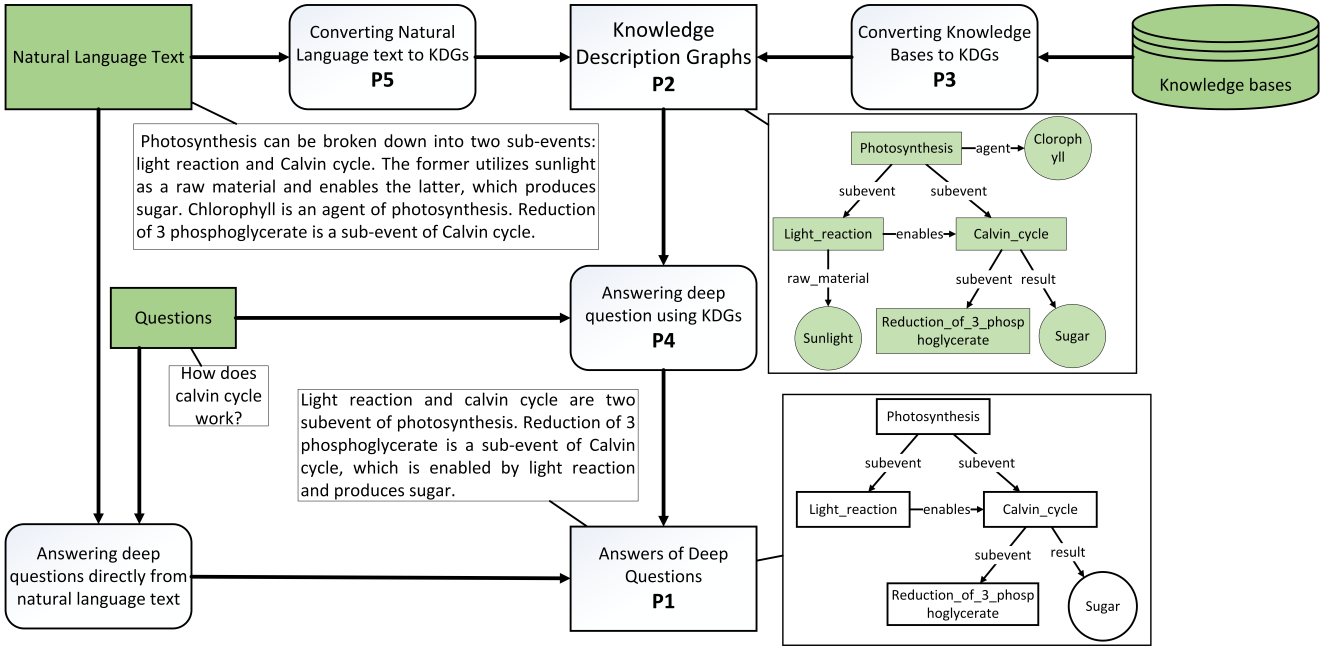

Figure 1.1: Steps in our deep question answering system.

Formulation:

(P1) Defining the answers of deep questions.

(P2) Defining Knowledge Description Graphs (KDGs).

Major tasks:

(P3) Constructing KDGs from existing knowledge bases.

(P4) Answering deep questions using KDGs.

(P5) Converting from natural language text to KDGs.

Chapter 2

ADDRESSING HOW QUESTIONS WITH RESPECT TO KNOWLEDGE BASES ABOUT EVENTS, ENTITIES, AND RELATIONSHIPS

2.1 Introduction

There exists a large body of knowledge bases about events, entities, and relationships. This includes many frame-based knowledge bases, such as the ones mentioned on Knowledge Machine's website [1] , ConceptNet (Liu and Singh, 2004; Speer and Havasi, 2013), and especially the knowledge base of AURA (Chaudhri *et al.*, 2009). Manually created, AURA's knowledge base represents knowledge in biology, chemistry, and physics. [2] It contains detailed information about events, entities, and the relationships among them. For example, regarding the biological process *photosynthesis*, it has information concerning the participants within this particular process, its sub-processes, such as *light reaction* and the *Calvin cycle*, its sub-processes' participants, the fact that *light reaction* enables the *Calvin cycle*, and many other relationships between the two components *in photosynthesis*. However, AURA addresses a limited set of factual question types, such as "What is photosynthesis?", "What are the similarities between prokaryotic and eukaryotic cells?", and "Which chemical bond is the agent of adhesion of water?"

It has been a dream as well as a challenge for the Question-Answering(QA) community to answer deeper questions, such as "Why" and "How," with respect to such knowledge bases and to text. For example, two "Deep KR&R Challenge Workshops" attempting this challenge were organized recently (DKRC2011, 2011; DKRC2012, 2012); in one of them, Chaudhri (Chaudhri *et al.*, 2012) lists several "How" and "Why" questions of interest.

[1] http://www.cs.utexas.edu/users/mfkb/related.html

[2] The AURA system used its knowledge based in passing AP tests (Gunning *et al.*, 2010).

"Why" and "How" questions have also been mentioned in the field of Question-Answering with respect to text (Higashinaka and Isozaki, 2008; Aouladomar, 2005; Verberne, 2009, 2006). However, they rely heavily on phrases such as "because of" and "causes of" to rank the set of retrieved documents that may contain the answers but do not provide direct ones. Such approaches are inadequate because the answer is either not explicitly mentioned, or parts of it are scattered throughout the text and only a part of it may be found by the QA systems.

According to the best of our knowledge, there is no research categorizing deep questions such as "How" and "Why" with respect to the answers they demand and no corpus has been created as such for evaluating such results. One of the most important factor behind this is the lack of well-founded formulations of deep questions i.e. the answer to *"What should the answer to such a question include?"*. Hence in our paper, we attempt to formulate such answers with respect to a knowledge base and pave an way for formalizing the answers to deep questions.

In this dissertation, we formulate the answers of "How" questions with respect to knowledge graphs. Our approach expands on the work of (Baral *et al.*, 2012b), where the authors attempted to formulate answers to two closely related questions: "How are X and Y related in process Z" and "Why is X important to Y?" with respect to a simpler graphical structure that they called "Event Description Graphs" (EDGs). Our approach is more general than the one mentioned in (Baral *et al.*, 2012b) in the following ways: (1) EDGs support a very limited set of nodes and edges while the Knowledge Description Graph structure, which we use in this work, is more general, and (2) our approach can be applied to a wider set of questions.

In a sense, formulating an answer to a "How" question is a deeply philosophical task. Thus, we explored the literature of such in various fields. In biology, Wouters (Wouters, 2005) identified several types of "How/Why" questions and expected answers. For exam-

ple, he mentions that "How is X used?" asks for the biological role/function and "How does X work" asks for physiological explanation. However, he does not give details about what should be included in the answers. In philosophy and the social sciences, Bunge's model (Bunge, 2004; Moss and Nicholson, 2012; Gerring, 2010) is often used to describe how a system works. It contains components, environmental items, and the ties between them, but lacks the details concerning answers to "How" questions.

Motivated by (DKRC2011, 2011; DKRC2012, 2012; Chaudhri *et al.*, 2011; Baral *et al.*, 2012b), we define Knowledge Description Graphs (KDGs) [3] by adapting "Object-Oriented" and frame-based representations from Knowledge Machine (KM) (Clark *et al.*, 2004) and AURA. Using KDGs, we represent the mechanism of "how" a process works and hence formulate a graph-based answer. We then expand the construct to answer more complex questions. Additionally, we apply our constructs on several examples of "How" questions and analyze their properties. Although we do not elaborate on it in this work, we have implemented the constructs formulated here to automatically answer "How" questions and reason about many aspects of biological mechanisms.

Besides the main contribution in formulating the answers to How questions, our work can also be extended to apply to QA systems which is based on some kind of knowledge graph (such as IBM Watson (Ferrucci *et al.*, 2010) and AURA (**?**)) to answer deeper questions, such as How questions.

In the following sections, we (1) propose a formal definition of KDGs, (2) formulate the answer of "How does X work?" using KDGs, and then (3) generalize the "How" questions' formulation to define answers to "How does X ρ Y?" where the ρ can be mechanism relations, like production, activation, and help. To make it easier to understand, we attempted to center our examples around the *plant* domain of biology. Our formalism is neither restricted to the plant domain nor to biology. Also to ease understanding, we provide the

[3] Please note that KDGs were first defined in a different context in a short paper.

textual descriptions of the answers in some examples, although those answers are actually in the form of graphs; we implemented a module to generate natural language answers from the these graphs. Note that those answers are in simple English and can vary from the textual description in the given examples.

2.2 Knowledge Description Graphs

A Knowledge Description Graph (KDG) (Baral and Vo, 2013) is a structure that represents the facts about instances and classes of events, entities, and the relationships between them. It is a directed graph structure with three types of nodes (**event**, **entity**, and **class** nodes) and five types of directed edges (**compositional**, **class**, **participant**, **locational**, and **ordering** edges). The first four types of edges are referred to as the main edges. KDGs are assumed to be directed acyclic graphs with respect to the main edges.

We used the slot names in KM (Clark *et al.*, 2004) and AURA as a guide to categorize the five types of edges. Figure 2.1 shows the types of edges in a KDG and its corresponding sources and destinations. For example, ordering edges must be from event to event while compositional edges must be from event to event or from entity to entity, depending on their specific relations. Table 2.1 summarizes the type of edges and their meanings. The first column contains the names of the edge types; the second column depicts the information the edges usually convey. For example, locational edge from *Src* to *Dest* means that the *Dest* (must be an event, see the Figure 2.1) happens in *Src* (must be an entity). The third column shows the example slot names in KM, corresponding to the edge type.

Each relation between *Src* and *Dest* in KM is described by a pair of slot names. For example, to tell that *Dest* is a part of *Src*, slots **has-part** and **is-part-of** are used. Slot **has-part** of *Src* has value *Dest* (*Src*[has-part] = *Dest*) and slot **is-part-of** of *Dest* (*Dest*[is-part-of] = *Src*). When the Object Graphs (Baral and Liang, 2012) are constructed from AURA, based on KM, the redundant slots were removed to simplify the Object Graph and

Edge type	Meaning	Relation(s)
locational	*Dest* happens in *Src*	happenings
class	*Src* is an instance of class *Dest*, or *Dest* is a super class of *Src*	instance-of, super-class
compositional	*Src* is usually composed of *Dest*	subevent, first-subevent, has-part, has-region, has-basic-structural-unit
ordering	there usually timely order between *Src* and *Dest*	next-event, enables, causes, prevents, inhibits
participant	entity *Src* is involved in event *Dest* as a participant, a result, etc.	raw-materials, result, agent, destination, instrument, origin, site

Table 2.1: Types of Edges in a KDG. An Edge (From *Src* to *Dest*) Usually Has the Meaning in the Second Column. The Third Column Contains the Example Slot Names in KM Corresponding to the Edge Type.

render them acyclic. This can be done by keeping the forward slot (such as **has-part**) and removing the reversed one (such as **is-part-of**) for most cases and using the reversed slot if the forward would create a cycle.

Following the Object Graphs, we assume that KDGs are directed acyclic graphs with respect to the main edges.

Figure 2.2 shows an example of a KDG. Three types of nodes, which are event, entity, and class, are respectively depicted by rectangles, circles, and hexagons. Ordering edges are represented by dashed-line arrows. All main edges are represented by solid lines.

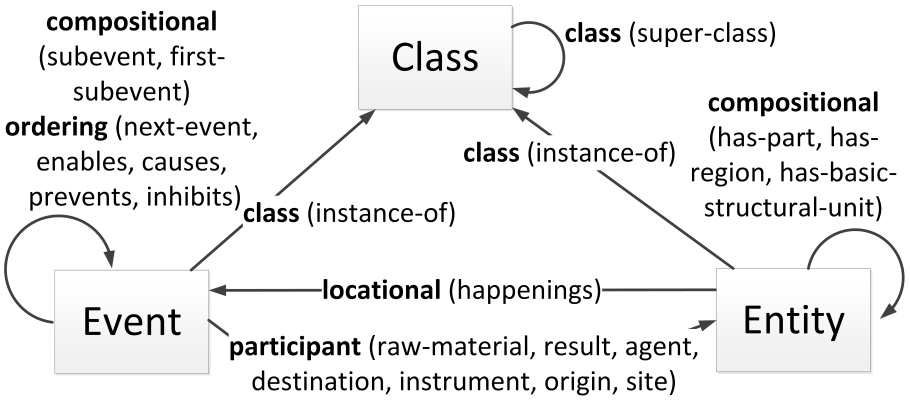

Figure 2.1: Types of edges in a KDG and the corresponding sources and destinations of the edges.

Compositional edges are represented by solid-line arrows, participant edges by lines with a black-square pointed to the entity node, class edges by diamond head arrows, and locational edges by lines with a circle pointed to the event node. Each event or entity node in a KDG has a specific name, like *Plant023* or *Photosynthesis342*, and has a edge to its class. However, for the sake of simplicity, we omit most of the class edges and class nodes from the examples, and we refer to event or entity nodes by their class names rather than specific names. For example, we shorten *Plant023* to *Plant* and *Photosynthesis342* to *Photosynthesis*.

Shown in Figure 2.2 and Figure 2.3 are two examples of KDGs: the KDG of *Plant* and the KDG of *Eukaryote*. The KDG of *Plant* (Figure 2.2) contains following information: (i) A plant cell is the basic structural unit of a plant; it contains chloroplast, where photosynthesis occurs, and (ii) Photosynthesis can be broken down into two sub-events: light reaction and the Calvin cycle. The former utilizes sunlight as a raw material and enables the latter, which produces sugar. Furthermore, Chlorophyll is an agent of photosynthesis. Reduction of 3 phosphoglycerate is a sub-event of the Calvin cycle.

Cpath and opath:

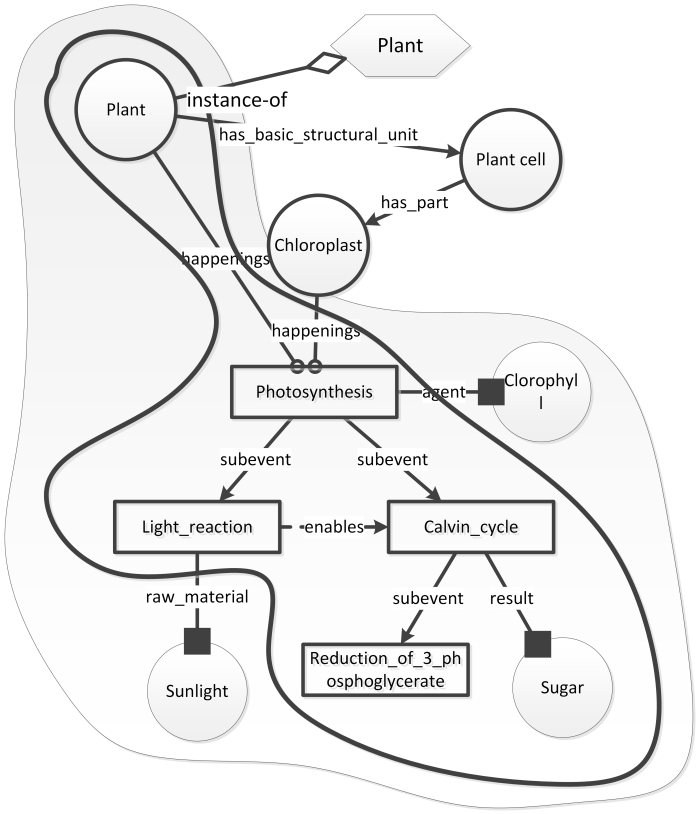

Figure 2.2: A KDG of *Plant*. An answer for "How does photosynthesis work?", a $HOW^{plant}(photosynthesis)$ (will be explained in the next section), is encircled by the light line. An answer for "How does Calvin cycle work?", a $HOW^{plant}(calcin\ cycle)$, is encircled by the bold line.

A $cpath$ in the KDG is a path consisting of only main edges. Similarly, an $opath$ consists of only ordering edges. The path from *Photosynthesis* node to *Light reaction* and to *Sunlight* in Figure 2.2 is a cpath. The path from *Light reaction* to *Calvin cycle* is an opath.

Knowledge Description Graph rooted at Z

If Z is node in KDG G, the **Knowledge Description Graph rooted at** Z (denoted as $KDG^G(Z)$) is the induced subgraph of G whose nodes are Z and all of Z's descendants

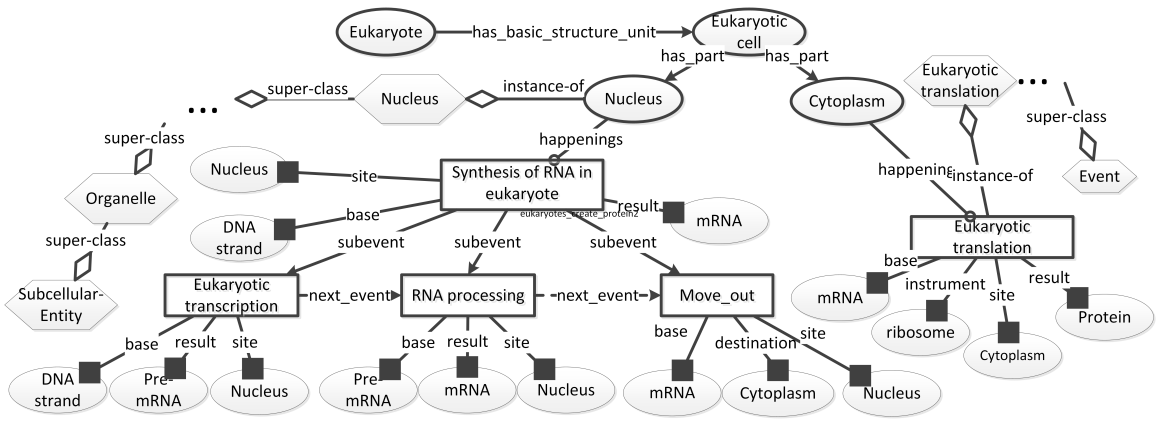

Figure 2.3: A KDG rooted at the entity *Eukaryote*.

through main edges (from now on, when mentioning the child and descendant in an KDG, we mean through the main edges). When G is clear from context, $KDG(Z)$ is used instead of $KDG^G(Z)$.

Lemma 1 *The descendant relation in KDG is a strict partial order.*

Proof 1 *We are going to prove descendant relation is a strict order by showing its irreflexivity, transitivity, and asymmetry. Given two nodes X and Y in a KDG, the descendant relation between them does not always exist. Thus, the descendant relation is a strict partial order.*

Irreflexivity: *Let X be a nodes in KDG G. X is not a descendant of X.*

Transitivity: *Let X, Y and Z be three nodes in KDG G. If X is a descendant of Y, Y is a descendant of Z, then X is a descendant of Z (by definition of descendant)*

Asymmetry: *Let Z and X be two nodes in a KDG G. If X is a descendant of Z then Z is not a descendant of X because there is no loop in G with respect to main edges.*

The following is an important proposition about the equality of the KDGs when they are obtained from different KDGs. For example, the KDG of *RNA processing* (Figure 2.3)

obtained directly from the KDG of *Eukaryote* is the same as the one obtained from the KDG of *Neucleus* (which was obtained from the KDG of *Eukaryote*).

Proposition 1 *Let X be a descendant of Z in a KDG G. Let $G1$ be the $KDG(X)$ obtained from G and $G2$ be the $KDG(X)$ obtained from $KDG(Z)$. We have $G1 = G2$.*

The proof of this proposition is given in the Appendix.

2.2.1 Highest Node - Lowest Node

Definition 1 *A property P of a node in the KDG is either characteristic of the node itself or of the entity/event/class it represents.*

Some example properties "produce mRNA," "is descendant of *Neucleus*," and "is a common ancestor of *Eukaryotic transcription* and *RNA processing*. The two nodes *RNA processing* and *Synthesis of RNA in eukaryote* in Figure 2.3 both satisfy the properties "has result mRNA" and "produce mRNA."

Depending on certain property P, its satisfiability need to be defined when it is not trivial. For example, an event has the property "has result mRNA" when it has the participant edge "result" to *mRNA*, but it also has the property when its subevent has "has result mRNA." These satisfiability conditions are not trivial and therefore need to be specified based on the property in used.

The definition of the "highest node" and "lowest node" that satisfies property P are given in the following.

Definition 2 *The highest nodes in the $KDG(X)$ that have property P - denoted as HI_P^X - are the maximal nodes with respect to the partial order "descendant" in lemma 1 that have property P. The lowest nodes in the $KDG(X)$ that have property P - denoted as LO_P^X - are the minimal nodes that have property P.*

In other words, node T is a "highest node" with property P if it possesses property P but none of its ancestor has property P.

Now, suppose there is a property $P1$ that *RNA processing*, *Synthesis of RNA in eukaryote* and *Eukaryotic translation* all satisfy (Figure 2.3). Note that *RNA processing* is a descendant of *Synthesis of RNA in eukaryote*. The highest nodes that have property $P1$ in $KDG(Eukaryote)$ are *Synthesis of RNA in eukaryote* and *Eukaryotic translation*, but the lowest nodes that have property $P1$ in $KDG(Eukaryote)$ are *RNA processing* and *Eukaryotic translation*.

Corollary 1 *$KDG(X)$ has at least one node that has property P. There exists at least one highest node and at least one lowest node in $KDG(X)$ that have property P.*

Proof 2 *The set of nodes in a $KDG(X)$ that have property P is a non-empty partial order set $\Longrightarrow$ there always exists maximal elements and minimal elements.*

Corollary 2 *Let X and Y be two nodes in $KDG(Z)$. There exists at least one lowest common ancestor of X and Y (denoted as $LCA^Z(X,Y)$). Let S be a set of nodes in $KDG(Z)$. There exists at least one lowest common ancestor of all the nodes in S (denoted as $LCA^Z(S)$).*

Proof 3 *Z is the ancestor of X and Y. Use Corollary 1, we have the result: there exists at least one lowest common ancestor of X and Y.*

2.3 The Answer to "How does X work?"

The answers to "How" questions are constructed from a KDG. If X is a node in the KDG, there are other nodes that can affect X (via incoming ordering edges). Also, there are nodes that are affected by X (via outgoing ordering edges). The answer is a graph, consisting of the nodes that affect X or are affected by X. The answer graph also contains edges as described below. Our answer construction is inspired by Bunge (Bunge, 2004),

which describes four attributes that include a set of "environmental" items. These items are those that affect or are affected by a node. We specify the environmental items via a KDG to arrive at our answer construction. Formally, the Environmental Set of X is the set $\{ N \mid X$ has an ordering edge to N OR N has an ordering edge to $X \}$. For example, in Figure 2.2, *Light reaction* and the *Calvin cycle* are in the environmental set of each other.

2.3.1 Contextual Path

In a KDG, if there is a cpath from node X to node Y, it implies that Y has a structural relationship to X. Such a cpath is called a contextual path of Y with respect to X. For example, a contextual path of *Photosynthesis* with respect to *Plant* is the path from *Plant* to *Plant cell*, to *Chloroplast* and then to *Photosynthesis*. It tells us that the *Chloroplast* is a part of *Plant cell*, the *Plant cell* is the basic structure of *Plant*, and *Photosynthesis* occurs in the *Chloroplast*.

The answer to "How" questions on X can be constructed by defining a subgraph consisting of nodes that are in the environmental set of X and the nodes that are in the contextual path of X with respect to the root of the KDG. To formalize, we first define the candidate subgraph, $HOW^Z(X)$.

Definition 3 ($HOW^Z(X)$) *Let X be a node in the $KDG(Z)$, a $HOW^Z(X)$ is a induced subgraph of $KDG(Z)$ whose set of nodes is the union of the following:*
(1) all the nodes in the $KDG(X)$; (2) all the environmental nodes of X; and (3) all the nodes on a contextual path of X (with respect to Z).

Shown in Figure 2.4 (b) is the visualization of $HOW^Z(X)$. The dark triangle is for the $KDG(X)$, the dashed lines to/from X are for X's environmental nodes and their edges from/to X, and the solid line from Z to X is for the contextual path of X with respect to Z. The other graphs are generalization of the "How does X work?" question and is explained in section 4.

Note that the question "How does X work?" is valid only if X is an event node. If X is an instance of *XClass*, then the real question is "How does XClass work?" instead of "How does X work?" However, we often ask this question of X rather than *XClass*.

Definition 4 (Answer to "How does XClass work?") *Let X be an event node in the* $KDG(Z)$ *and X be an instance of XClass. An answer of the question "How does XClass work?" is a* $HOW^Z(X)$.

Using $HOW^Z(X)$, we can picture the answer of "How does photosynthesis work?" in Figure 2.2. Since no event affects or is affected by photosynthesis, the answer is the subgraph of the KDG rooted at *Photosynthesis* and the path to it from *Plant*. Similarly, because the *Calvin cycle* is affected by the *Light reaction*, the answer to "How does the Calvin cycle work?" contains not only the KDG rooted at the *Calvin cycle* but also at the node *Light reaction* and the path to *Calvin cycle* from *Plant*.

If the question has a scope such as "How does X work in T?," given $KDG(Z)$, the answer would be $HOW^T(X)$ where T is used instead of Z as a root node.

2.3.2 Justifying Our Formulation of $\mathrm{HOW}^{\mathrm{Z}}(\mathrm{X})$

For an answer such as $HOW^Z(X)$ or latter structures, we justify their correctness by three ways:

1. Conceptually, by their components: The graphical structure must contain all concepts that the answer should have according to our common sense. For example, $HOW^Z(X)$ contains all concepts mentioned in Bunge's model.

2. By examples: We apply the structure to a specific question and show that it give us an "intuitive" answer. We did this in the example "How does photosynthesis work?" in the previous section.

3. By their properties: The graphical structure should also satisfy important properties such as existence, uniqueness, and inclusiveness. The inclusiveness property is that the

structure of answer A should contain the structure of answer B when answer A contains answer B. For example, conceptually, the answer of "How does photosynthesis work?" should contain the answer of "How does the Calvin cycle work?" because the *Calvin cycle* is a sub-event of *photosynthesis*.

As the first two ways of justifying $HOW^Z(X)$ were shown previously, we justify the formulation of $HOW^Z(X)$ by its properties.

Lemma 2 (Existence, uniqueness) *Let X be a node in the $KDG(Z)$, there exists at least one $HOW^Z(X)$. Each of them differs from the others by the contextual path of X.*

To relate $HOW^Z(X)$ and $HOW^Z(C)$ (for the inclusiveness) where C is a descendant of X in the $KDG(Z)$, we define a notion of "self-contained." The rationale behind the self-contained condition is that the encoding of each event should contain only information that is needed to describe it. Often when the self-contained condition is not satisfied, then there is a knowledge encoding error where the same event is incorrectly used multiple times. This can be corrected by creating extra copies of that event and linking them appropriately. For example, the events *Light reaction* and *Calvin cycle* describe the event *Photosynthesis*, and there should not be the event *Preparation* which enables *Light reaction* when it itself is not a child of *Photosynthesis*. If the annotators wanted to encode that *Preparation* enables event *Light reaction* in some other circumstance, they should have created another event similar to *Photosynthesis* and included *Preparation* as its sub-event.

The self-contained condition is formally defined as follows:

Definition 5 (Self-contained) *A node X is self-contained if one of the following conditions is satisfied.*

- *X does not have any children.*
- *All of X's children are self-contained and their environmental nodes (if they exist) are also X's children.*

$KDG(Z)$ is self-contained if all its nodes are self-contained.

According to the definition, all of the nodes in Figure 2.2 are self-contained and the $KDG(Plant)$ is self-contained. For example, all the children of *Light reaction* and *Calvin cycle* are self-contained because they do not have children. The environmental nodes of *Light reaction* and *Calvin cycle* are among themselves and are children of *Photosynthesis*. Thus, *Photosynthesis* is also self-contained. Moreover, from the definition we can easily show that if C' is an environmental node of C and C is a node in self-contained $KDG(X)$, then C' must be also in $KDG(X)$. If the KDG is self-contained, then a $HOW^Z(X)$ constructed from it has the inclusiveness property: $HOW^Z(X)$ should contain $HOW^Z(C)$ if X is an ancestor of C. More formally,

Corollary 3 *Let C be a descendant of X in $KDG(Z)$, C' is an environmental node of C. If $KDG(X)$ is self-contained, C' is also a descendant of X.*

Proposition 2 (Inclusiveness) *Let X be a node in the $KDG(Z)$ where $KDG(Z)$ is self-contained. Let C be a descendant node of X through cpath. Each $HOW^Z(X)$ contains at least one $HOW^Z(C)$.*

The proposition above shows that the answer of "How does photosynthesis work?" contains the answer to "How does the Calvin cycle work?" (Figure 2.2) since the *Calvin cycle* is a descendant of *Photosynthesis*.

In this proposition, $HOW^Z(X)$ does not necessarily contain all $HOW^Z(C)$ because there could be a path from Z to C that does not pass through X.

Proof 4 *Let $G1$ be a $HOW^Z(X)$. We are constructing a subgraph $G2$ of $G1$ that is a $HOW^Z(C)$.*

By definition, $HOW^Z(X)$ contains $KDG(X)$, which, in turn, contains $KDG(C)$.

*Let P*1 *and P*2 *be two cpaths in G*1. *P*1 *is from Z to X, and P*2 *is from X to C. Let P be the concatenation of P*1 *and P*2.

*Construct the induced subgraph G*2 *of G*1 *that contains*

1. *all the nodes in KDG*(C),

2. *all the nodes on P,*

3. *all C's environmental nodes in KDG*(Z),

Using Corollary 3 we have: all C's environmental nodes in KDG(X) *and, thus, also in G*2. *Hence, G*2 *is a* $HOW^Z(C)$.

2.3.3 Algorithm to Construct $\mathrm{HOW}^{\mathrm{Z}}(\mathrm{X})$

We can construct the $HOW^Z(X)$ through three steps similar to those below. First, we start from Z and search for X, traveling through only main edges. Once we reach X, we have the cpath from Z to X. After that we can obtain X's environmental nodes, which are connected to/from X through behavioral edges. $KDG(X)$, the last component of $HOW^Z(X)$, can be obtained by searching all the reachable nodes from X through main edges.

```
INPUT: KDG(Z), X
OUTPUT: HOW^Z(X)
-------------------------
STEP 1: Search for X in KDG(Z) and get the cpath from Z to X
STEP 2: Search for X's environmental nodes
STEP 3: Get all nodes that X can reach through main edges
```

Since a KDG is Directed Acyclic Graph (DAG) with respect to main edges, all the steps above can be done in linear time and linear space (Sedgewick and Wayne, 2011). i.e. its complexity is $C*(V+E)$ where V and E are the number of nodes and edges in the *KDG*.

$HOW^Z(X)$, thus, can be constructed by algorithms of linear time and space complexity.

2.4 Generalization - "How does X ρ Y?"

In this section, we formulate answers to questions of the form "How does X ρ Y?." They generalize the answer formulation in the previous section; the describing event now needs to satisfy some additional constraints.

2.4.1 Types of Procedural Questions

We analyzed more than 300 "How" questions from the collection of 1,800 biology questions and answers [4] , and we found that almost all the questions fall into the following three types with some slight variations about the scope of the question. i.e. "X of Z" or "X in Z" instead of only "X" or "Y."

1. How does *X* reproduce/grow/appear/... ?
2. How is *Y* activated/formed/treated/produced/... ?
3. How does *X* create/help/participate-in/... *Y*?

All three types can be generalized to a single type, "How does X ρ Y?", where ρ is a mechanism relation such as "help," "participation," and "creation," upon others. If *X* or *Y* is missing, the general type becomes types 1 and 2, as listed above. We now construct answers to these generalized questions.

2.4.2 The Generalized Model 1

The answer to "How does X ρ Y?" should contain structures to show (1) the structural relation between *X* and *Y*; (2) the relation ρ from *X* to *Y*, and (3) details on how *X* works. Part (1) can be represented with three cpaths: the contextual path to a lowest common ancestor *LCA* of *X* and *Y* and the two contextual paths of *X* and *Y* with respect to the *LCA*.

[4] http://www.biology-questions-and-answers.com/

[5] The three cpaths are joined at *LCA* like an upside-down "Y" (see Figure 2.4 2.4(a)) with the three end-points X, Y and Z. Part (3) can be represented by $HOW^Z(X)$. We need to construct the structure for (2) expressing the relation ρ from X to Y. That structure is an undirected path from X to Y, called the relational path.

Definition 6 (Relational path) *Let X and Y be two different nodes in $KDG(Z)$. A relational path from X to Y (denoted as $REL^Z(X,Y)$ is an **undirected** path of $KDG(Z)$ from X to Y through any types of edges.*

Examples of some relational paths shown in Figure 2.5 are:

1. $REL^{Photosynthesis}(sunlight, sugar)$ is the union of cpath from *Light reaction* to *Sunlight*, cpath from *Calvin cycle* to *Sugar* and opath from *Light reaction* to *Calvin cycle* (highlighted by solid line in Figure 2.5)

2. $REL^{Photosynthesis}(sunlight, sugar)$ is the union of cpath from *Photosynthesis* to *Sunlight* and cpath from *Photosynthesis* to *Sugar*. It is highlighted by the dashed line in Figure 2.5.

If the information conveyed by the nodes and edges along the path shows that $X\ \rho\ Y$, we said that it explains the relation ρ. For example, the path from *Photosynthesis* to *Calvin cycle* to *Sugar* shows that *Photosynthesis* produces *Sugar*, therefore explaining the relation "produce." Similarly, the path from *Light reaction* to *Calvin cycle* explains the relation "enable" or "activate." Identifying the precise properties that the nodes and edges on the path (explaining a relation ρ) should satisfy and automatically matching them will be discussed in a future work. Here, we loosely define the relational path explaining relation ρ as follows:

[5]Note that in a $KDG(Z)$, two nodes X and Y always have at least an *LCA* since Z is their common ancestor. However, *LCA* is not unique and our definitions account for that non-uniqueness.

Definition 7 (Relational path explaining relation ρ) *A relational path from X to Y explains the relation ρ if the information conveyed by the nodes and edges along the path shows that X ρ Y.*

Armed with the previous definitions, we can construct $HOW1^Z(X,\rho,Y)$ as follows.

Definition 8 ($HOW1^Z(X,\rho,Y)$) *Let X and Y be two nodes in $KDG(Z)$, LCA be a lowest common ancestor of X and Y. A $HOW1^Z(X,\rho,Y)$ is the union of the following subgraphs:*

- *Cpaths from LCA to X, from LCA to Y and from Z to LCA (contextual path of LCA with respect to Z),*
- *a relational path from X to Y explaining relation ρ, and*
- *the $KDG(X)$ and edges between X and its environmental nodes.*

Definition 9 *Let X and Y be two nodes in $KDG(Z)$ and be instances of XClass and YClass respectively. A $HOW1^Z(X,\rho,Y)$ is an answer of the question "How does XClass ρ YClass?"* [6]

The problem of answering question "How does X ρ Y?" now becomes the problem of finding the relational path explaining ρ using the semantics of edges and nodes along the path (We give examples in the next subsection).

Depending on the level of details and the specific relation ρ, the $HOW1^Z(X,\rho,Y)$ structure may have some slight modifications such as including more relational path(s) or excluding the $KDG(X)$.

[6] There could be many answers corresponding to different *LCA*s.

2.5 Algorithm to Construct HOW1^Z(X, ρ, Y)

We can construct the $HOW1^Z(X,\rho,Y))$ through three steps similar to those below. First we begin from Z and search for X, traveling through only main edges. Once we reach X, we have the cpath from Z to X. After that, we can obtain X's environmental nodes, which are connected to/from X through behavioral edges. $KDG(X)$, the last component of $HOW^Z(X)$, can be obtained by searching all the reachable nodes from X through main edges.

```
INPUT: KDG(Z), X
OUTPUT: HOW^Z(X)
-------------------------
STEP 1: Search for X and Y in KDG(Z)
STEP 2: Search for lowest common ancestor LCA of X and Y; obtaining
   cpath from Z to LCA, cpaths from LCA to X and LCA to Y
STEP 3: Search for X's environmental nodes and KDG(X)
STEP 4: Search for relational path R from X to Y
```

Since a KDG is a Directed Acyclic Graph (DAG) with respect to main edges, steps 1-3 can be done in linear time and space (Sedgewick and Wayne, 2011). If step 4 needs a more complex algorithm, its complexity will be the complexity of the whole algorithm in constructing $HOW1^Z(X,\rho,Y))$. The algorithm to find relational paths can be found only when the precise conditions of the nodes and edges of the path explaining a relation are determined.

In the following examples, we discuss the complexity in specific cases where these conditions are determined.

2.5.1 Examples

In this section, we will consider some examples of applying $HOW1^Z(X,\rho,Y)$ to specific questions. In each question, we will analyze how $HOW1^Z(X,\rho,Y)$ changes and verify

the information it gives.

"How does X work?"

In this case, X and Y of $HOW1^Z(X,\rho,Y)$ are combined into one and there is no relation ρ. The three cpaths become the contextual path from Z to X. Hence, $HOW1^Z(X,\rho,Y)$ (Figure 2.4 (a)) becomes $HOW^Z(X)$ (Figure 2.4 (b)) as expected.

"How does X produce Y?"

In this case, there must exist a relational path ρ explaining relation "produce" from X to Y. Let us assume that conditions of "X produces Y" are defined as follows.

Definition 10 *Let X and Y be two nodes in $KDG(Z)$.*

- *X directly produces Y if there is a participant edge "result" from X to Y, and*
- *X produces Y if X directly produces Y or X's component directly produces Y.*

Translating to the conditions of the relational paths: A relational path ρ explaining the relation "produce" must be a cpath from X to Y where the last edge is the participant edge "result". Because the relational path from X to Y must be a cpath, Y actually belongs to $KDG(X)$, and $HOW1^Z(X,produce,Y)$ also becomes the $HOW^Z(X)$ as shown in Figure 2.4 (c).

Based on the above, the highlighted path in the Figure 2.6 explains the relation "produce": *Plant* "produces" *sugar*, as *Photosynthesis* and *Calvin cycle* do. The answer to "How does photosynthesis in plants produce sugar?", pictured in Figure 2.6, can be read as follows:

Example 1 *Q: How does plant produce sugar?*
A: "Plant" is where photosynthesis occurs. Photosynthesis has two sub-events: light reaction and Calvin cycle. The light reaction enables the Calvin cycle, which produces sugar. Reduction of 3 phosphoglycerate is a sub-event of Calvin cycle.

Algorithm to construct $HOW1^Z(X,Y)$ When we sort $KDG(Z)$ in the topological order, all the nodes on any cpath from X to Y must lie between X and Y. To find a relational path ρ explaining relation "produce" from X to Y, we first search for a node N in between X and Y in topological order that has the participant edge "result" to Y. We then find an arbitrary cpath from X to Y. All these two steps can be done in linear time and space.

As shown above, the other parts of $HOW1^Z(X,Y)$ can also be constructed in linear time and space. Hence, $HOW1^Z(X,Y)$ can be constructed in linear time and space.

"How does X participate in Y?"

In a KDG, an entity C directly participates in an event B if there is a participate edge from B to C. However, if B is a sub-event of A, C also participates in A. The relational path that explains the relation "participate" is thus an undirected path starting from X following a participant edge to P_1 (the directed participant edge is from P_1 to X), and then continuing on the compositional edges to P_2, P_3,..., and then to Y.

Note that the participant edge is from event to entity, P_1 must be event node; "How does X participate in Y?" only makes sense when X is an entity and Y is an event. If X is a subevent of Y, X does not participate in Y because X is already a "part of" event Y".

We formally define this "participant" relational path as follows.

Definition 11 *Let X be an entity node and Y be an event node in $KDG(Z)$. $PPATH^Z_{X,Y}$ is a relational path in $KDG(Z)$ that contains the nodes $X, P_1, P_2, ..., P_n$ ($n \geq 1$) where: $P_n = Y$, there exists a participant edge from P_1 to X and a compositional edge from P_{i+1} to P_i ($1 \leq i \leq n-1$).*

In the following, there are various questions and answers concerning the "participate" relation in $KDG(photosynthesis)$. Again, depending on the detail level of the answer, some parts of the $HOW1^Z(X,Y)$ can be skipped within the answer. For example, the short answers of those questions can contain only the relational paths.

Figure 2.7 shows the answers in the following examples with respect to the *KDG* of photosynthesis. The answers of the questions "How does sunlight participate in light reaction?" and "How does sunlight participate in photosynthesis?" are $PPATH^{photosynthesis}_{sunlight,light\ reaction}$ and $PPATH^{photosynthesis}_{sunlight,photosynthesis}$ respectively. Note that according to our definition, sunlight does not participate in the Calvin cycle or "'Plant"; it only participates in light reaction and photosynthesis. The answers in the graph can be translated to natural language like the answers in the examples using the techniques discussed in section 2.8.

Example 2 *Q: How does sunlight participate in light reaction?*
A: Sunlight is the raw material of light reaction.

Example 3 *Q: How does sunlight participate in photosynthesis?*
A: Sunlight is the raw material of light reaction, which is a subevent of photosynthesis.

Example 4 *Q: How does sunlight participate in reduction_of_3_phosphoglycerate?*
A: N/A. This question does not have an answer since there is no evidence of sunlight participate in reduction_of_3_phosphoglycerate.

The following lemma relates the $PPATH^{Z}_{X,Y'}$ and $PPATH^{Z}_{X,Y}$ when Y is ancestor event of Y' (i.e., there is a cpath from Y to Y' through compositional edges of sub-event relation). For example, the $PPATH^{photosynthesis}_{sunlight,photosynthesis}$ contains $PPATH^{photosynthesis}_{sunlight,light\ reaction}$ as shown in Figure 2.7.

Lemma 3 (Inclusiveness of PPATH) *Let X be an entity node in $KDG(Z)$, Y and Y' be two event nodes in $KDG(Z)$ where Y is an ancestor event of Y'. If $PPATH^{Z}_{X,Y'}$ exists, there is a $PPATH^{Z}_{X,Y}$ that contain a $PPATH^{Z}_{X,Y'}$. In other words, if X participates in Y', it also participates in Y.*

Proof 5 *The needed $PPATH^Z_{X,Y}$ can be constructed by concatenating $PPATH^Z_{X,Y'}$ and subevent path between Y' and Y'. Moreover, when X only participate in Y through Y' (i.e. all participant path from X to Y go through Y'), all $PPATH^Z_{X,Y}$ contain $PPATH^Z_{X,Y'}$.*

Once we find the satisfied relational path, the $HOW1^Z(X, \rho, Y)$ can be obtained. Here, because of semantics of the question, Y can not be a descendant of X, and $KDG(X)$ is not a must-have component of the answer; hence, we can remove it from the answer, leaving us the structure in Figure 2.4 (d).

"How is X used for Y?"/"How can X help Y?"/ "How does X act in Y?"

These questions are closely related to the question "How does X participate in Y?" where X is an entity and Y is an event. They imply that X is directly involved in Y or that X is indirectly involved in Y. We believe that the question "How is X used for Y?" is more likely in the case of "directly involved" (which can be answered similarly to "How does X participates in Y?") and the rest are closer to the case of "indirectly involved."

Here we consider one of the "indirectly involved" questions according to the following assumption. We assume that "X helps Y" implies that all of the following conditions are satisfied:

1. $X = P_1$ or X participates in P_1
2. $P1$ directly "helps" $P2$ because there is an opath (the path contains only ordering edges) from $P1$ to $P2$
3. $P_2 = Y$ or $P2$ participates in Y

Formally, we define the relational path $HPATH^Z_{X,Y}$ explaining relation "help" as follows.

Definition 12 *Let X be an entity node and Y be an event node in KDG(Z). $HPATH^{Z}_{X,Y}$ is a relational from X to Y that contains: an optional $PPATH^{Z}_{X,P_1}$, opath from P_1 to P_2 and an optional $PPATH^{Z}_{P_2,Y}$. The two PPATHs are optional.*

Similar to previous question, the $KDG(X)$ can removed from the answer of "How can X help Y?," leaving us the structure in Figure 2.4 (d).

Example 5 *Q: How does sunlight help calvin cycle?*
A: Photosynthesis has two subevents: light reaction and calvin cycle. Sunlight is a raw material of the light reaction. The light reaction enables the calvin cycle

Figure 2.8 shows the answer in the example above with respect to $KDG(photosynthesis)$. It contains a $HPATH^{photosynthesis}_{sunlight,calvin\ cycle}$ which goes from the *sunlight* to the *Calvin cycle* through the *light reaction*. Since the answer is with respect to $KDG(photosynthesis)$, the root Z of the $HOW1^{Z}(X,\rho Y)$ is *photosynthesis*. *Photosynthesis* is also a LCA of *sunlight* and *Calvin cycle*. The three cpaths of $HOW1^{Z}(X,\rho Y)$ becomes two: cpaths from *photosynthesis* to *sunlight* and to *Calvin cycle*.

"How is Y activated in Z?"

This question is the passive form of "How does X activate Y in Z?", where X is missing. We must first find X in $KDG(Z)$ that "activates" Y and then obtain results similar to that of the active case shown in Figure 2.4 (e).

Assume that the relational path explaining "activate" is an opath ended by ordering edge "enable". The opath from *light reaction* to *Calvin cycle* explains the relation "activate". Thus, Figure 2.8 shows the answer of "How is the Calvin cycle activated in photosynthesis?". This is the complete answer including the $KDG(X)$ of $HOW1^{Z}(X,\rho,Y)$, which is the $KDG(light_reaction)$ in this example.

2.5.2 *Justifying* HOW1^Z(X, ρ, Y)

As we mentioned before, we justify the $HOW1^Z(X,\rho,Y)$ by three ways. The first two (conceptual and example) were shown in the previous subsection. Here we give justification by showing the properties of $HOW1^Z(X,\rho,Y)$.

Lemma 4 (Existence, uniqueness) *Let X and Y be two nodes in KDG(Z). If there exists a relational path from X to Y that explains ρ, there exists at least one $HOW1^Z(X,\rho,Y)$.*

Moreover, since $HOW1^Z(X,\rho,Y)$ is the generalized answer, we expect it to transform to the $HOW^Z(X)$, similarly to the examples in the previous section.

Proof 6 (Proof by construction) *Given X and Y in KDG(Z). There exists at least one lowest common ancestor LCA of X and Y. Since the Z is the ancestor of LCA, LCA is ancestor of X and Y, there exists at least one cpath from Z to LCA, one cpath from LCA to X, and one cpath from LCA to Y. Using the given relational path from X to Y that explains ρ with the three cpaths above, the KDG(X), and X's environmental nodes, we can construct a $HOW1^Z(X,\rho,Y)$.*

Lemma 5 *Let X and Y be two nodes in KDG(Z). If Y and other nodes in the relational path of $HOW1^Z(X,\rho,Y)$ are descendants of X, then $HOW1^Z(X,\rho,Y)$ becomes $HOW^Z(X)$.*

Proof 7 *If Y and other nodes in the relational path of $HOW1^Z(X,\rho,Y)$ are descendants of X, X is LCA of X and Y. KDG(X) obviously contain cpath from LCA to X and LCA to Y. All components of $HOW1^Z(X,\rho,Y)$ construct a $HOW^Z(X)$.*

However, $HOW1^Z(X,\rho,Y)$ does not satisfy the inclusiveness property, which motivates the alternative answer structure $HOW2^Z(X,\rho,Y)$ defined in the following section.

2.5.3 The Generalized Answer 2

An alternative formulation of the answer to the question "How does X ρ Y?" could be a $HOW^Z(T)$ structure, where T is chosen in such a way that the answer would contain the relational path ρ. To keep the answer as compact as possible, a possible choice of T is a lowest common ancestor of all the nodes in the relational path.

Definition 13 ($HOW2^Z(X,\rho,Y)$) *Let X and Y be two nodes in $KDG(Z)$. Let REL be a relational path from X to Y explaining the relation ρ. A $HOW2^Z(X,\rho,Y)$ is $HOW^Z(LCA)$ where LCA is a lowest common ancestor of all the nodes in REL.*

Definition 14 *Let X and Y be two nodes in $KDG(Z)$, X and Y are instances of XClass and YClass. A $HOW2^Z(X,\rho,Y)$ is an answer of the question "How does XClass ρ YClass?".*

Figure 2.4 (f) illustrates $HOW2^Z(X,\rho,Y)$; the shaded area is for the $KDG(LCA)$. As we can see, $HOW2^Z(X,\rho,Y)$ contains more information than $HOW1^Z(X,\rho,Y)$. So for a specific question and the level of detail we want, we can remove some part of it from the answer; I.e. the shaded area above line ρ or below line ρ or $KDG(Y)$.

2.5.4 Justifying $\mathrm{HOW2^Z(X,\rho,Y)}$

$HOW2^Z(X,\rho,Y)$ obviously satisfies the existence, uniqueness, and inclusiveness properties of $HOW^Z(X)$. For specific questions, it also collapses to simpler structures similarly to $HOW1^Z(X,\rho,Y)$. Moreover, $HOW2^Z(X,\rho,Y)$ has an additional inclusiveness property stated below.

Lemma 6 (Inclusiveness) *Let $HOW2^Z(X,\rho,Y) = HOW^Z(LCA)$ be the structure in $KDG(Z)$ answering question "How does X ρ Y?". $KDG(Z)$ is self-constained. Let T be a node on the relational path of $HOW2^Z(X,\rho,Y)$ so that the relational path from T to Y still explains the relation ρ. The $HOW2^Z(X,\rho,Y)$ answering "How does X ρ Y?" contains at least one $HOW2^Z(T,\rho,Y)$, the structure answering question "How does T ρ Y?"*

Proof 8 *Let LCA*1 *a lowest common ancestor of the relational path p*1 *from X to Y; LCA*2 *be a lowest common ancestor of the relational path p*2 *from T to Y. Given p*1 *contains p*2*, we only need to prove that the* $HOW^Z(LCA1)$ *contains* $HOW^Z(LCA2)$*. Since p*1 *contains p*2*, LCA*1 *must be ancestor of LCA*2*. There are two possible cases:*

1. *If LCA*1 = *LCA*2*:* $HOW^Z(LCA1) = HOW^Z(LCA2)$*, thus* $HOW^Z(LCA1)$ *contains* $HOW^Z(LCA2)$*.*

2. *If LCA*1 $\neq$ *LCA*2*: A* $HOW^Z(LCA1)$ *contains at least one* $HOW^Z(LCA2)$ *(Proposition 2)*

2.6 Cascading the $\text{HOW1}^{\text{Z}}(\text{X}, \rho, \text{Y})$ and $\text{HOW2}^{\text{Z}}(\text{X}, \rho, \text{Y})$

In this section, we very briefly discuss how to use $HOW1^Z(X, \rho, Y)$ to answer more complex "How" questions. Combinations of multiple $HOW2^Z(X, \rho, Y)$ can be done in a similar manner.

Consider the question "How does X enable Y to create T in Z?". The answer to this question contains two smaller ones: *how X enables Y in Z* and *how Y creates T in Z* joint at Y. Compared to two separated structures, the combined answer shown in Figure 2.9 gives us a better representation of the entire process. For example, it allows us to causally reason about X and T, i.e. figure out what would happen to T if X is missing or malfunctioning.

2.7 Answering Other Types of Questions

In this section, we are going to describe how the structures we defined in previous sections can be used to answer other types of question.

2.7.1 Answering "Why is X important to Y (in Z)?" / "Why is X needed for Y (in Z)?"

"Why is X important to Y?" is a very popular "Why" question, as observed in the collection of 1,800 questions on http://www.biology-questions-and-answers.com/. We defined the answer of this question using EDG (Baral *et al.*, 2012b). Because of the limited EDG

structure, the actual question answered is "Why is X important to Y in event Z?" where both X and Y must be in event Z. In the following, using the KDG, which is more general than EDG, we redefine the answer of the general question "Why is X important to Y (in Z)?" where X, Y and Z can be either events or entities.

When X plays some role in Y, the question could be answered by explaining the importance of X's role in Y. This approach is similar to the answered of "Why does entity X require entity Y?" discussed in (Chaudhri *et al.*, 2012). When X does not directly play any role in Y, one possible answer of this question may have the following pattern:

1. X is important to P_1 because ... (explain the relation of X and P_1)
2. P_1 is important to P_2 because ...
3. ...
4. P_{k-1} is important to P_k because ...
5. P_k is important to Y because ....
6. Thus X is thus important to Y.

The importance of X to P_1, P_{k-1} to P_k or P_k to Y is justified by the relations or edges in KDG between them. X is important to Y because of the chain of "important" relations. In the example 6 is the model answer of "Why is sunlight important to photosynthesis?" following this pattern.

Example 6 *Question: Why is sunlight important to photosynthesis?*
Model answer: Sunlight is the raw material of light reaction thus sunlight is important to light reaction, light reaction is an important sub-event of photosynthesis, therefore sunlight is also important for photosynthesis.

Edge type	Relations	Direction	Meaning
ordering	all relation	same direction	*Src* is important to *Dest*
compositional	subevent	reversed direction	*Dest* is important to *Src*
compositional	subevent	reversed direction	*Dest* is important to *Src*
participant	raw-material, agent	reversed direction	*Dest* is important to *Src*
participant	result	same direction	*Src* is important to *Dest*
...			

Table 2.2: Some Relations Corresponding to "Important" Edges. If the Edge Are From *Src* to *Dest*, the Third Column Indicates if *Src* Is Important to *Dest* (Same Direction) or *Dest* Is Important to *Src* (Reversed Direction).

Graphically, this answer is the answer of "How is X important to Y?" which contains a relation path from X to Y through P_i ($1 \leq i \leq k$) explaining relation "important". Each edge from A to B of this relational path must explain relation "important" from A to B.

Let us take the point that B is important to A; if without B, A does not function properly, (A is an event) or A's existence/completeness is affected (A is an entity).

The specific types of edges explaining the "important" relation can vary according to the KBs in use. Using our KB (based on KM), we assume that an ordering edge from A to B indicates that the "important" relation has the same direction as the edge (A is important to B); other types may have the reversed direction (2.2). Once the important relations are defined, they become special edges between two nodes. Figure 2.10 shows the $KDG(photosynthesis)$ with these "important" edges. The "important" relation from *sunlight* to *light reaction* is represented by the double-head arrow in reversed direction of the participant edge "raw-material" from *light reaction* to *sunlight*.

The "important" edges in Figure 2.10 can be read as:

1. *sunlight* is the raw material of *light reaction* thus *sunlight* is important to *light reaction*.

2. *light reaction* is important to *calvin cycle* because the former enable the latter.

3. sub-events like *light reaction* and *calvin cycle* are important to *photosynthesis*.

4. *calvin cycle* is important to *sugar* because it produces *sugar*.

With the "important" edges, finding the relational path explaining the "important" relation from X to Y becomes finding possible path(s) from X to Y through only "important" edges. For example, there are two possible such paths from *sunlight* to *photosynthesis*. One path only goes through *light reaction*; the other goes through both *light reaction* and *Calvin cycle*.

Once we have a relational path explaining the relation "important," the answer of "Why is X important to Y in Z?" can be constructed using $HOW1^Z(X, important, Y)$ or $HOW2^Z(X, important, Y)$ as the formal definition below. The short answer again can skip the parts unrelated to the relational path. Figure 2.10 shows the relational path explaining why *sunlight* is important to *photosynthesis*, which is the answer to "Why is sunlight important to photosynthesis?" with respect to $KDG(photosynthesis)$.

Definition 15 *Let X and Y be two nodes in $KDG(Z)$. The answer of the question "Why is X important to Y in Z?" is a $HOW1^Z(X, important, Y)$ or $HOW2^Z(X, important, Y)$.*

2.7.2 Answering the Question "How are X and Y related?"

In the technical communication (Baral *et al.*, 2012b), we only considered Event Description Graphs (EDG) to answer questions of the form "How are X and Y related in M?," where M was restricted to events, as the EDG does not have edges from entities to events. However, KDGs allow such edges and thus can be used to answer questions where M could be either an event or an entity.

Given the KDG of entity 1 in Figure 2.11, let us consider the question "How are entity 8 and event 5 related?". Conventional wisdom has it that the answer should only contain important information to understand the relation between 8 and 12 such as:

1. What is event 2 with respect to entity 1? where event 2 is the of entity 8 and event 5
2. What is entity 8 with respect to event 2?
3. What is entity 12 with respect to event 2?
4. The behavioral relation between 8 and 12? For example, does entity 8 help in enabling or disable entity 5?

Following this intuition, we define the answer for question "How are X and Y related?," named $MIN\_KDG^{Z}_{X,Y}$, as another subgraph of the KDG. The $MIN\_KDG^{Z}_{X,Y}$ is a variation of the $HOW1^{Z}(X,Y)$ as illustrated in Figure 2.12. It also contains three cpaths: from Z to a lowest common ancestor LCA of X and Y, from LCA to X, and from LCA to Y. However, since this question concentrates on the relations, we would include more relations between X and Y. For instance, we would include all possible opaths between the last two cpaths above (from LCA to X and from LCA to Y). Combining with the segments in the two cpaths, these opaths will becomes various relational paths from X to Y that contain at least one ordering edge showing behavioral relations from X to Y.

We formally define $MIN\_KDG^{Z}_{X,Y}$ as follows.

Definition 16 *Let X and Y be two nodes in an $KDG(Z)$. $MIN\_KDG^{Z}_{X,Y}$ is an induced subgraph of $KDG(Z)$ consisting of the nodes on the following paths:*

1. *A contextual path of $LCA(X,Y)$ with respect to Z*
2. *A contextual path V_x of X with respect to $LCA(X,Y)$*
3. *A contextual path V_y of Y with respect to $LCA(X,Y)$*

4. *All opaths from X' to Y' where $X' \in V_x$, and $Y' \in V_y$*

The first three paths above explain the structural relation between X and Y while the last one explain the behavioral (or ordering) relation from X to Y. In the example in Figure 2.11, this path is from event 3 to event 5 through 4.

Example 7 *Q: How are sunlight and sugar related in plants?*
A: Photosynthesis is a biological process that occurs in plant. Photosynthesis has two subevents: light reaction and calvin cycle. The light reaction needs sunlight as its raw material, and later enables the calvin cycle which produces sugar.

Figure 2.13 shows the answer in the example 7 $MIN\_KDG^{plant}_{sunlight,sugar}$. We have that $LCA(sunlight, sugar)$ in $KDG(plant)$ is *photosynthesis* and follows paths from definition 16:

1. A contextual path of $LCA(sunlight, sugar) = photosynthesis$ with respect to *Plant* contains *plant*, *plant cell*, *chloroplast* and *photosynthesis*
2. A contextual path V_x of *sunlight* with respect to *photosynthesis* contains *photosynthesis*, *light reaction*, and *sunlight*.
3. A contextual path V_y of *sugar* with respect to *photosynthesis* contains *photosynthesis*, *calvin cycle* and *sugar*
4. All opaths from X' to Y' where $X' \in V_x$, and $Y' \in V_y$: $\{\{light\_reaction, calvin\_cycle\}\}$

The entity *Clorophyll* and the event *Reduction of 3 phosphoglycerate* are excluded from the answer because they are not important to understand the relation between sunlight and sugar.

Properties of MIN_KDG

Similar to HOW1, the MIN_KDG has the existence and uniqueness properties but not the inclusiveness.

Lemma 7 (Existence, uniqueness) *Let X and Y be two nodes in the $KDG(Z)$, there always exists a $MIN\_KDG^{Z}_{X,Y}$.*

In special cases when X or Y is descendant of the other, the ancestor will become a lowest common ancestor of X and Y. The MIN_KDG now becomes the cpath from Z go through both X and Y.

Lemma 8 *Let X and Y be two nodes in the $KDG(Z)$. When Y is a descendant of X, a cpath from Z to X and to Y is a $MIN\_KDG^{Z}_{X,Y}$.*

2.7.3 Discuss About Answering "Why does X create Y?"

Moldovan (Moldovan *et al.*, 2000) used a single answer type *reason* for all "Why" questions. Based on the types of adverbial clauses in (Quirk *et al.*, 1985), Verberne (Verberne, 2006) categorized *reason* into four subtypes based on their semantic: cause, motivation, circumstance, and purpose. In the following are examples of these four subtypes from (Verberne, 2006).

Example 8 (Cause)

Q: Why did the flowers get dry?

A: The flowers got dry because it hadn't rained in a month.

Example 9 (Motivation)

Q: Why do you water the flowers?

A: I water the flowers because I don't like to see them dry.

Example 10 (Circumstance)

Q: Why should we be able to finish this today?

A: Seeing that it is only three, we should be able to finish this today.

Example 11 (Purpose)

Q: Why do people have eyebrows?

A: People have eyebrows to prevent sweat running into their eyes.

Obviously, one does not necessary have all four types of an answer. For example, cause and purpose make more sense in answering "Why does plant produce sugar?". The answer of the first type could be the description of the chain of processes leading to the creation of sugar. The answer of the second type could be a description of the importance of creating sugar to plant (i.e. plants create sugar to store energy, which an important process needed for plant.)

The information needed to answer the other two subtypes are not usually encoded in our KB; thus, we will only discuss cause and purpose.

Cause: We can answer the question by describing the chain of processes leading to creating Y. For example, for question "How does plant produce sugar?" is the *Light reaction* enabling *Calvin cycle*, which creates sugar. This answer is similar to the answer of "How does plant produce sugar?": $HOW1^Z(X, create, Y)$ or $HOW2^Z(X, create, Y)$. If process B, which directly makes the action (such as producing sugar), is caused by process A, A would be included in the answer:

1. If X is the process B, $HOW1^Z(X, \rho, Y)$ or $HOW2^Z(X, \rho, Y)$ would contains A because it is the environmental node of X.

2. If B is descendant of X, A would be in the $KDG(X)$ if the self-contained condition is satisfied. (See Definition 5)

Motivation: One possible answer could be description of the importance of creating sugar to plant (i.e. plants create sugar to store energy, which an important process needed for plant). The answer in this case is similar to the answer of "Why is Y important to X?"

2.8 Generate Answers in Natural Language

The answers in KDGs can be translated to natural language using the simple templates. We are using SimpleNLG (Gatt and Reiter, 2009) package to generate simple sentences from KDG. In the following is the ASP facts representing a KDG in "has( )" format mentioned earlier. Each line has a tuple of three things: $(X, Relation, Y)$. We define the sentence template for each type of relation and substitute X and Y to generate output sentences. For example, '"X is an instance of Y, Z and T." is the template for the 3 following tuples.

```
has(X, instance_of, Y).
has(X, instance_of, Z).
has(X, instance_of, T).
```

In the following are a KDG answering question "How does Calvin Cycle work?" and its representation in ASP.

```
% _Photosynthesis1514
event(photosynthesis1514).
has(photosynthesis1514, subevent, light_reaction13426).
has(photosynthesis1514, subevent, calvin_cycle13425).

% _Light-Reaction13426
has(light_reaction13426, enables, calvin_cycle13425).

% _Calvin-Cycle13425
has(calvin_cycle13425, result, sugar13435).
has(calvin_cycle13425, subevent, reduction_of_3_phosphoglycerate6918
    ).
```

From those facts, we can generate the following sentences:

```
Sugar13435 is a result of calvin_cycle13425.
Reduction_of_3_phosphoglycerate6918 is a subevent of
    calvin_cycle13425.
```

```
Calvin_cycle13425 and light_reaction13426 are subevents of
   photosynthesis1514.
Light_reaction13426 enables calvin_cycle13425.
```

2.9 Conclusions

Recent research has brought Question Answering to a much larger audience. However, most of the research has focused on factual questions. In this work, we investigated some cases of the more involved "Why" and "How" questions. We take the view that answers to such questions need to be initially formulated with respect to structured knowledge before basing them on free text, and hence we introduced the notion of KDG. The KDG is an abstract structure that contains information about objects and events such as classes, components, properties, sub-events, and the relationship between events and objects.

Our goal in this work was to formulate answers of "How" questions. We showed that the answer construction can be derived from a Knowledge Description Graph (KDG) by extracting the environment set and the contextual path corresponding to an entity or event X. From this result, we generalized our method to answer questions like "How does X ρ Y?," where X is a subject of ρ (i.e. ρ is a relation such as reproduction, growth) or X is an object of ρ (i.e. ρ is a relation such as activation, formation, production). We can also cascade answers to two or more X ρ Y questions to answer more complex procedural (or mechanism) questions. We justified our answer formulations by showing that they satisfy several natural properties and also through examples. Subsequently, we showed that the proposed structures can also be used to answer other "Why" and "How" such as "Why is X important to Y?", "How are X and Y related?", and "Why does X ρ Y?".

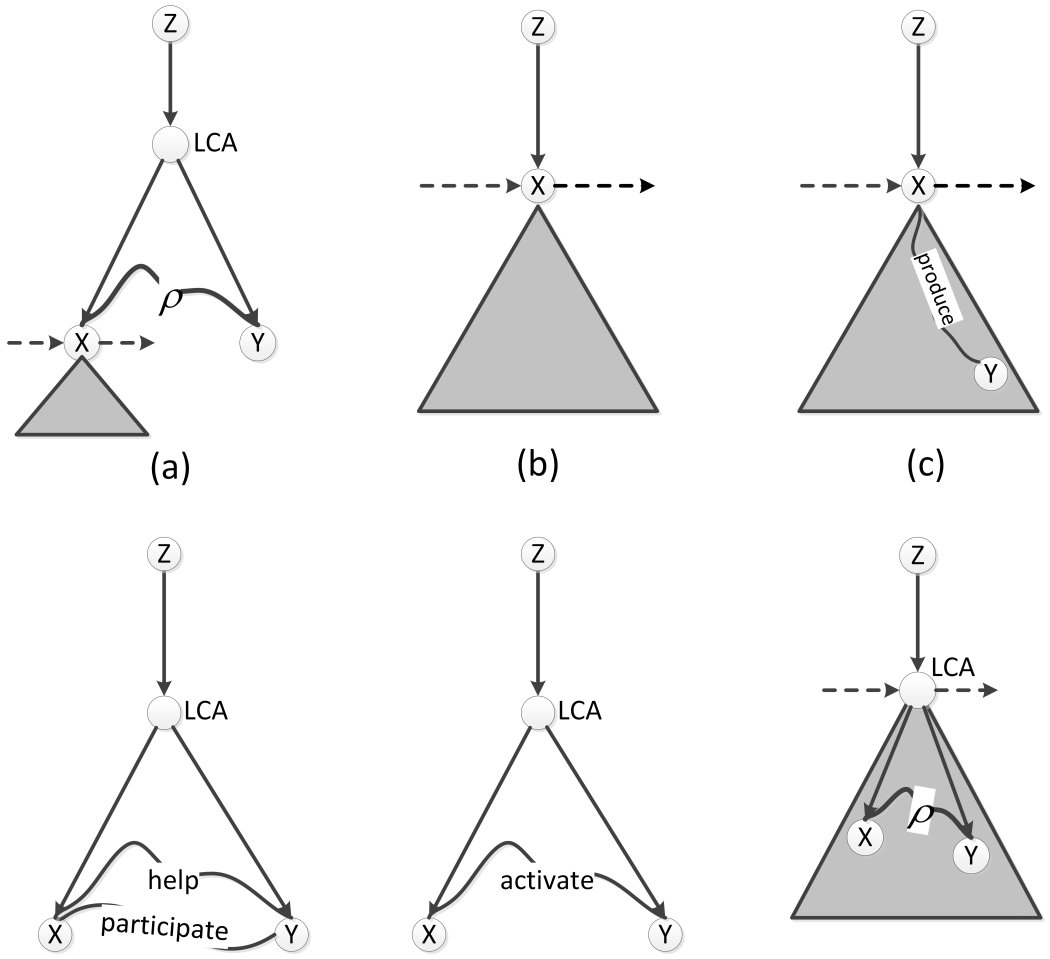

Figure 2.4: Visualization of the answers to various "How" questions:
(a) $HOW1^Z(X,\rho,Y)$ answering "How does X ρ Y?" This generic answer can be transformed into the answers of simpler questions (in subfigure (b)-(d)). For example, for question "How does X produce Y?" path ρ becomes *produce* and Y must be in $KDG(X)$, subfigure (a) become (c).
(b) $HOW^Z(X)$ answering "How does X work?"; The dark triangle is for the $KDG(X)$, the dashed lines to/from X are for X's environmental nodes and their edges from/to X, and the solid line from Z to X is for the contextual path of X with respect to Z.
(c) $HOW1^Z(X,produce,Y)$ answering "How does X produce Y?";
(d) $HOW1^Z(X,\rho,Y)$ answering "How does X help/participate/activate in Y?";
(e) $HOW2^Z(X,\rho,Y)$ answering "How does X ρ Y?," another generic answer to question "How does X ρ Y?". It gives more details than the one in subfigure (a) and satisfies more properties (of a "proper" answer form our intuition) such as inclusiveness

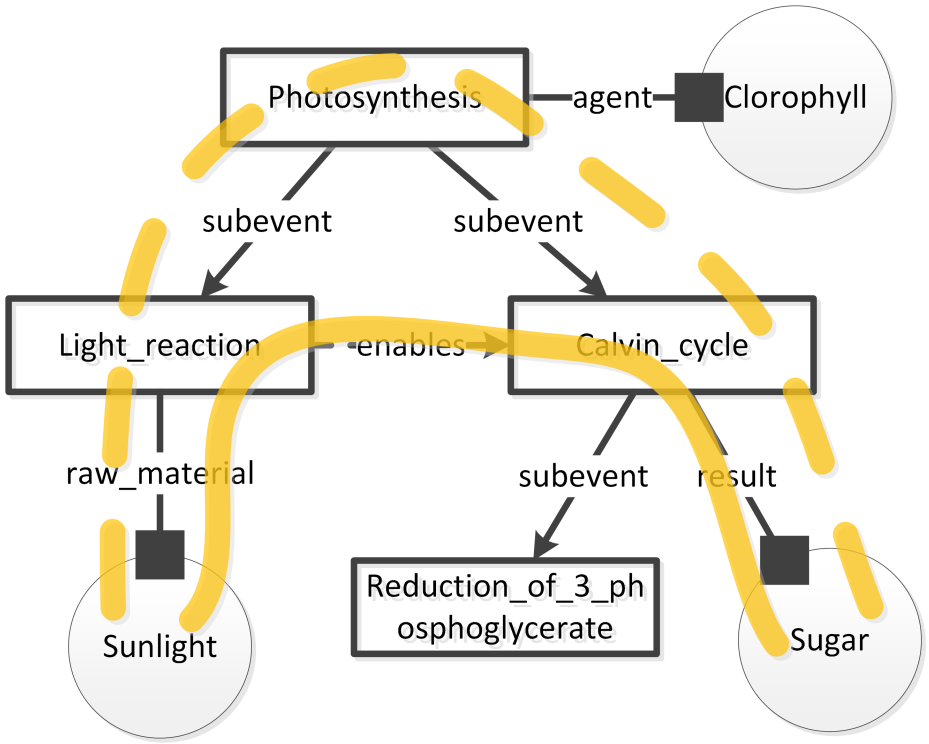

Figure 2.5: Some relational paths in KDG of Photosynthesis

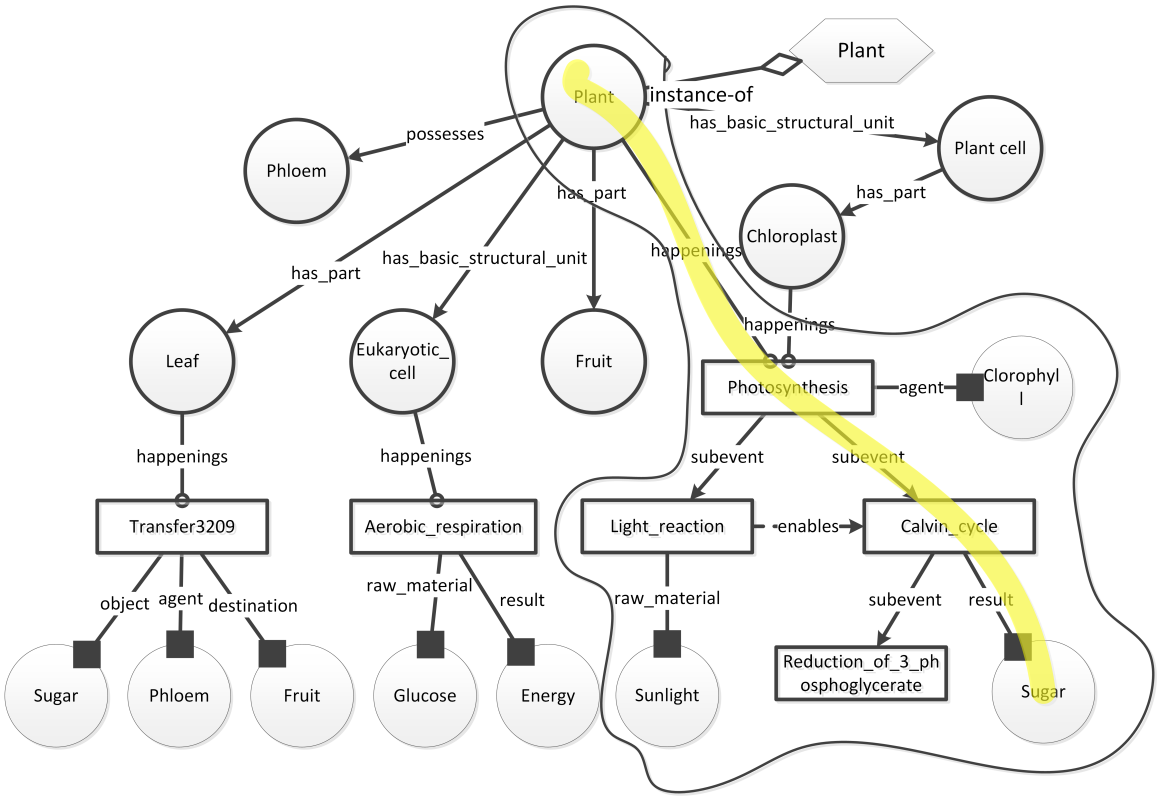

Figure 2.6: The answer of question "How does plant produce sugar?," $(plant, photosynthesis)\_KDG$, is enclosed by solid line. The highlighted path from *Plant* to *Sugar* explains relation "produce"

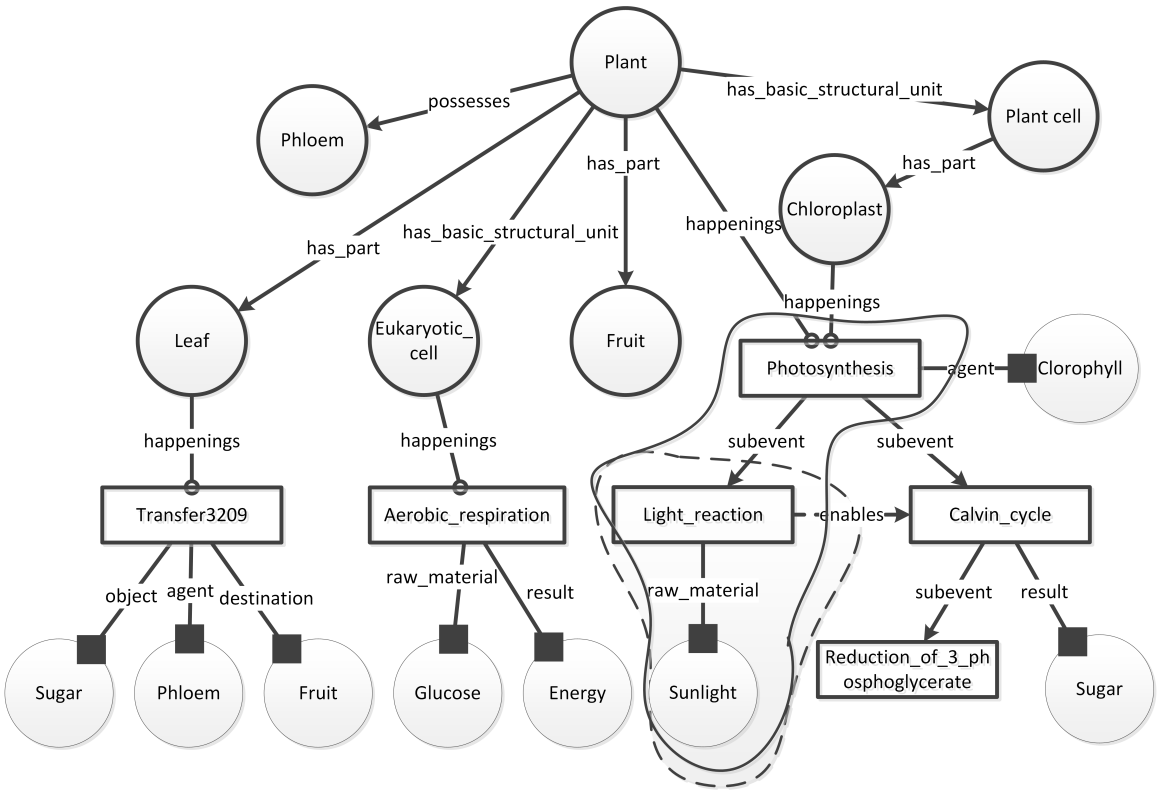

Figure 2.7: Various relational paths explaining relation "participate" in $KDG(photosynthesis)$:

(a) $PPATH^{photosynthesis}_{sunlight,light\ reaction}$, explaining how *sunlight* participates in *light reaction*, is surrounded by a dashed enclosure.

(b) $PPATH^{photosynthesis}_{sunlight,photosynthesis}$, explaining how *sunlight* participates in *photosynthesis*, is surrounded by a solid enclosure.

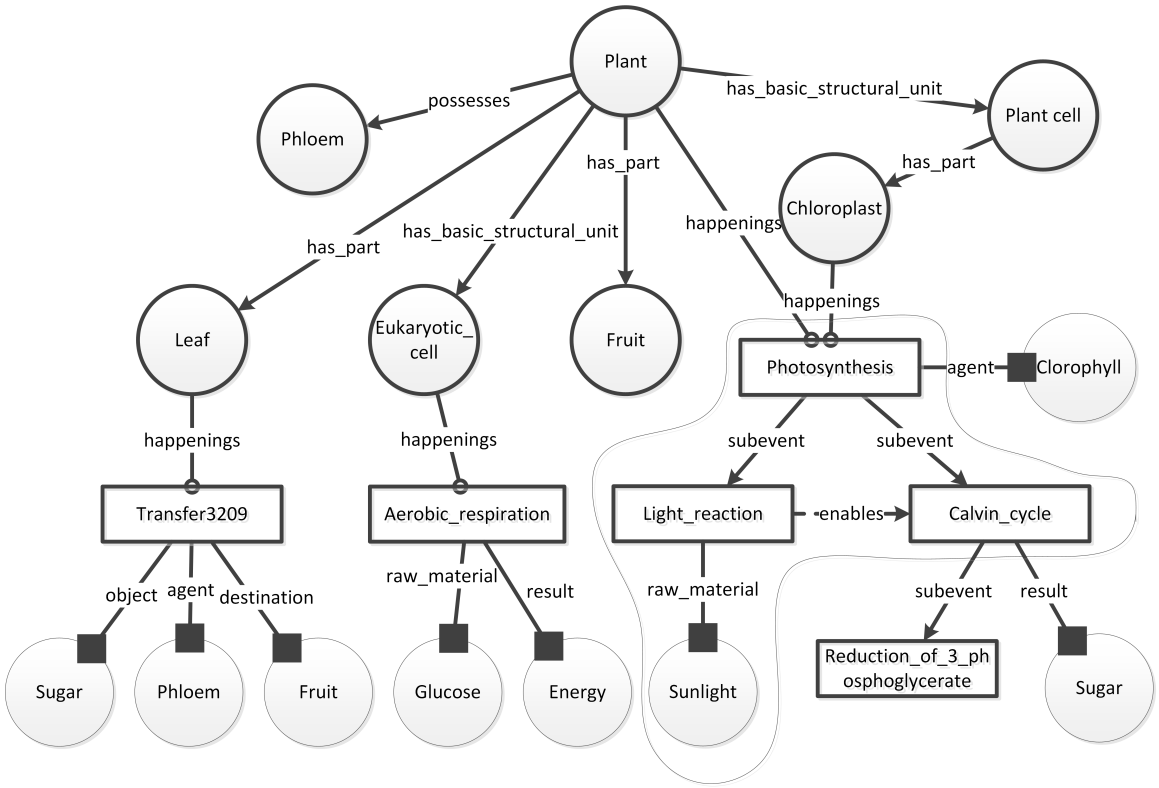

Figure 2.8: *The answer of the question "How does sunlight help calvin cycle (in photosynthesis)?" in KDG(photosynthesis) is enclosed by solid line. Coincidently, it is also the answer of "How is calvin cyle activated in photosynthesis?"*

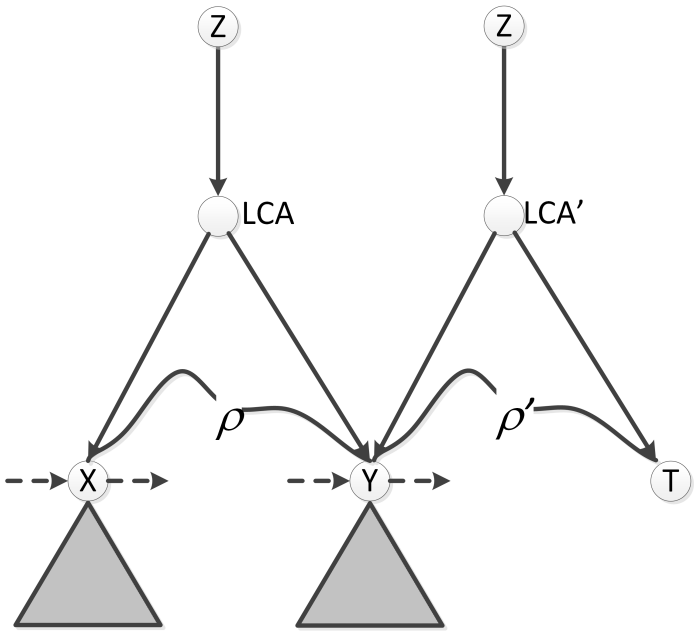

Figure 2.9: The combination of $HOW1^Z(X,\rho,Y)$ and $HOW1^Z(Y,\rho',T)$ joint at Y

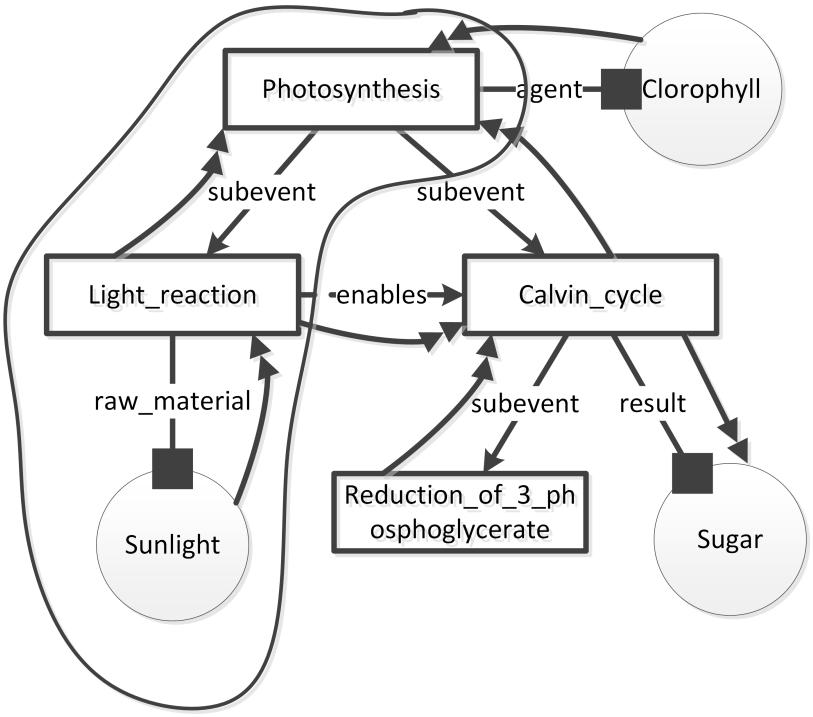

Figure 2.10: "Important" edges are represented by double-head arrows. A relational path explaining relation "important" from *sunlight* to *photosynthesis* is surrounded by the solid enclosure.

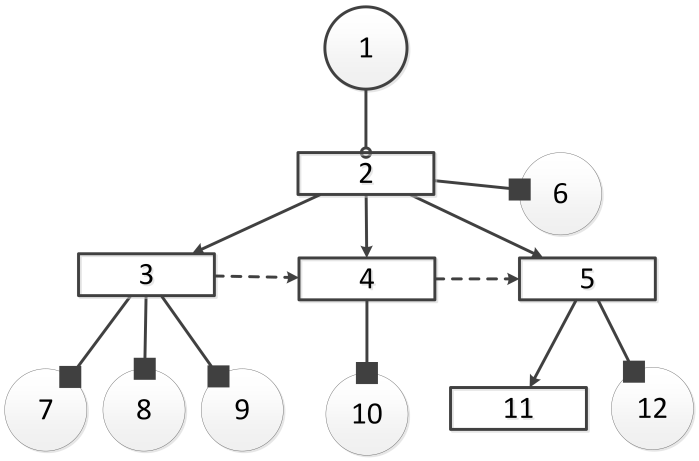

Figure 2.11: The KDG of entity 1

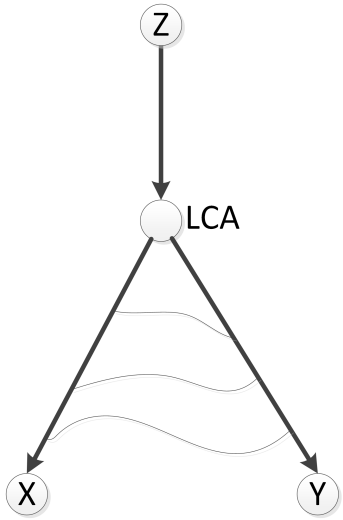

Figure 2.12: Visualization of $MIN\_KDG^{Z}_{X,Y}$.

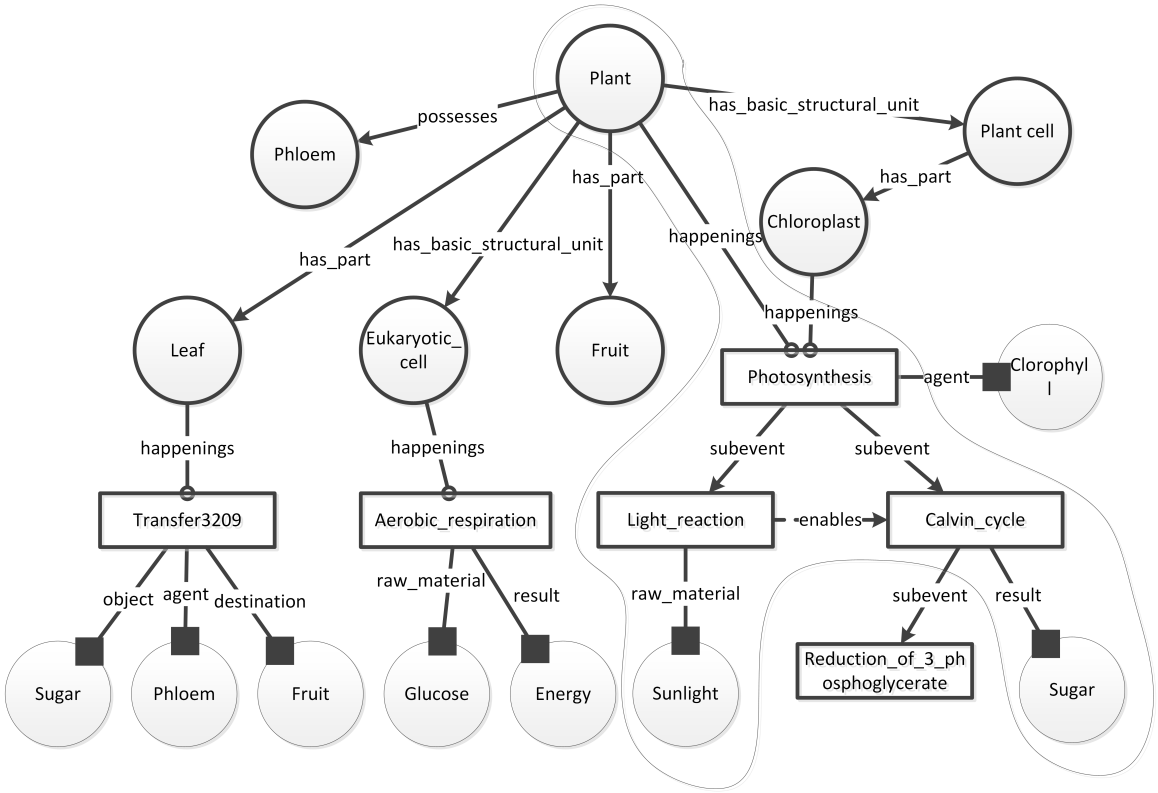

Figure 2.13: The answer of the question “How are sunlight and sugar related in plant?”

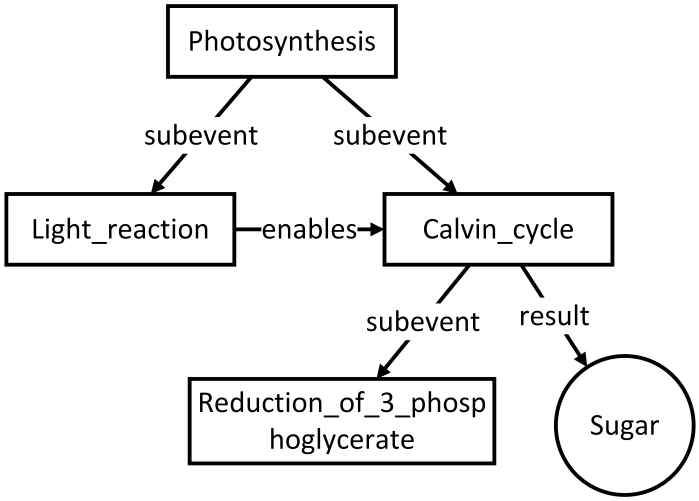

Figure 2.14: A KDG answering question “How does Calvin Cycle work?”

Chapter 3

ANSWER SET PROGRAMMING BASED REPRESENTATION AND REASONING WITH RESPECT TO FRAME-BASED KNOWLEDGE BASES

3.1 Introduction

A large body of such KBs have been developed using frame-based representations (Fikes and Kehler, 1985) [1] . The University of Texas hosts a website [2] listing a large number of these KBs, such as AURA (Chaudhri *et al.*, 2009), Cyc (Matuszek *et al.*, 2006), FreeBase (Bollacker *et al.*, 2008), WordNet (Miller, 1995), EcoCyc (Karp *et al.*, 2002), RiboWeb (Altman *et al.*, 1999), and others. While "object," "instance," and "oriented" terms arised during the late 1950s and early 1960's at MIT and in 1960-61 by Sutherland, the first frame-based knowledge representation (KR) is often considered to appear in 1975 (Minsky, 1975).

Knowledge in frame-based KR is organized in frames. Each frame has slots which can be filled with values or other frames. Similarly to object-oriented structure, there are hierarchical ordering frames which determine whether one frame may inherit the properties of another. Because of these inbuilt features, i.e. inheritance, frames allow concise representation of knowledge. For example, to describe a Sphynx cat, one can specify its unique features (i.e. its lack of a coat) in the set of frames corresponding to Sphynx cat class. One do not need to list other common features that it inherits from cat family, mammal, or animal.

Over the years, there has been some work to logically formalize the inheritance aspect of frames (Touretzky, 1986; Horty, 1994; Brewka, 1987). However, those work do not

[1] See also http://www.cs.man.ac.uk/∼stevensr/onto/node14.html

[2] http://www.cs.utexas.edu/∼mfkb/related.html

address "cloning" and "unification," the procedural reasoning mechanisms in many frame-based systems. The cloning operation allows an object (i.e. a new hybrid breed of cat named Blabla) to copy all the properties of another object (i.e. Sphynx cat), while Blabla is not necessary be one subtype of Sphynx cat. Unification operation combines existing properties of an object with all possible properties that it can inherit (through inheritance and cloning), then resolves all the conflicts and yields the complete information about that object. For example, unification on Sphynx cat will acquire additional information from Cat, Mammal, Animal, and others; combine with existing information in Sphynx cat to have a complete view about Sphynx cat. Unification needs to do more than well-known overriding and overloading operations in Object-Oriented programming. For example, Sphynx cat and normal cat respectively have two different objects *Skin_Sphynx* and *Skin_Cat*. Running unification on Sphynx cat is not simply choose one of the two types of skin, but merging two types into one unified type, and possibly repeat the merging process further (i.e. merging properties of the skin).

The first attempts to formalize cloning and unification were introduced in a parallel fashion in (Baral and Liang, 2012) and (Chaudhri and Son, 2012). Although both were inspired from the unification procedure in the frame-based systems KM (Clark and Porter, 2011) [3] , and both use AURA [4] , they are still substantially different from each other. For example, while (Chaudhri and Son, 2012) is more clear about the principles of unification, (Baral and Liang, 2012) covers more cases of unifications and how unifications can be applied recursively. Inspired by our previous work (Baral and Liang, 2012), we propose different and more comprehensive formalization of cloning and unification. By capturing the context of cloning and unification, it is better in specifying the conditions and how to recursively accomplish unification; it has the strength of both (Baral and Liang, 2012) and

[3] KM has been adopted by various projects, notably Project Halo (Chaudhri *et al.*, 2009).

[4] AURA is also developed under Project Halo.

(Chaudhri and Son, 2012). More details about the comparison among the four systems are discussed in section 3.8.

After giving the formal definition of unification using cloning and inheritance, we give an Answer Set Programming (ASP) based implementation of it. Afterwards, we show how to use ASP to answer several different types of factual questions. These questions were supported in AURA and discussed in (Baral and Liang, 2012). In this work, we revise and change the answering in (Baral and Liang, 2012) with respect our new formalism. In the process, we also suggest the optimization for ASP rules and mention its impact with respect to a large biological KB such as AURA.

3.2 Background

3.2.1 Answer Set Programs

We chose Answer Set Programming (ASP) (Gelfond and Lifschitz, 1988) as our knowledge representation language because of the following reasons:

1. ASP has simple syntax yet is expressive; it is non-monotonic, which is required in our non-monotonic formalism of unification and cloning.

2. ASP has a strong theoretical foundation with many building-block results (Baral, 2003), allowing us to prove the correctness of our implementation in this work.

3. It has several efficient solvers: (Gebser *et al.*, 2008; Niemelä and Simons, 1997; Leone *et al.*, 2006), allowing us to scale our solution to large KB as AURA.

An ASP program is a collection of rules of the form:

$$a \leftarrow a_1, \ldots, a_m, \mathit{not}\ a_{m+1}, \ldots, \mathit{not}\ a_{m+n}$$

where a, $a_1,\ldots,a_{m+n}$ are atoms. The rule reads as "a is true if $a_1 \ldots a_m$ are all known to be true and $a_{m+1} \ldots a_{m+n}$ can be assumed to be false". The semantics of answer set

programs are defined using answer sets. An entailment relation ($\models$) with respect to answer set programs is defined as follows: a program Π entails an atom p iff p is true in all the answer sets of Π.

3.2.2 Frame-based Knowledge Representation and Reasoning

Similar to (Baral and Liang, 2012) and (Chaudhri and Son, 2012), we also choose KM as our reference frame-based formalism. Knowledge in KM is organized in frames; each frame has slots which can be filled with values or other frames. Usually, KM's slots specify properties of class/instance or relations between them. KM provides the structured representation for classes and instance via slots in the following format, where "superclasses" and "instance-of" are two special meaning slots for representing class and instance.

```
(every <class> has
  (superclasses (<superclass1> ...
       <superclassN>))
  (<slot1> (<expr11> <expr12>...))
  (<slot2> (<expr21> <expr22>...))...)

(<instance> has
  (instance-of (<class1>...<classn>))
  (<slot1> (<expr11> <expr12>...))
  (<slot2> (<expr21> <expr22>...))...)
```

Some key slots that have special meanings for representation and reasoning in KM are listed below:

- **Superclasses:** The slot "superclasses" specifies the parent class of a class. All the superclass information in the KB form a hierarchy for all the concepts in the KB.
- **Prototype-of:** This slot defines the prototype(s) of a class. A prototype is an instance that serves as a basis of a class. If c is a prototype of class C, all instances

of *C* should copy all slots values (properties) of *c*, unless restricted otherwise via a specified prototype scope. How exactly these slot values are copied is defined as a unification process and will be formally defined later.

- **Instance-of:** This slot defines the class that an instance belongs to. A prototype *c* of class *C* is also an instance of *C*. If *a* is an instance of class *C*, then *a* gets a copy of all the slot values of *C*'s prototype.

- **Cloned-From:** This is a shortcut in KM. If an instance *a* has this slot with value *b* (read as *a* is cloned from *b*), then all *b*'s slot values are copied to *a*'s.

Users can interact with KM through queries. KM uses deterministic construction rules (Clark and Porter, 2011) to provide answers to queries. Both the queries and rules are normally based on instances; KM has several specialized reasoning modules to capture inheritance, cloning, and unification.

In the following, we illustrate KM's reasoning process with respect to an example about *Vehicle* which is slightly modified from the example in (Clark and Porter, 2011):

```
(Vehicle1 has
  (instance-of (Vehicle))
  (prototype-of (Vehicle))
  (has-engine ((a Engine with
      (strength (*Powerful))
      (fuel ((a Gas-type with
        (combustibility (*Hi)))))))))

(every Car has
   (superclasses (Vehicle)))

(Car1 has
   (instance-of (Car))
```

```
(prototype-of (Car))
(has-engine ((a Engine with
   (size (*Average))
   (fuel ((a Gas-type with
     (type (*Unleaded)))))))))
```

In the example, there are two classes (*Vehicle*, *Car*) and two instances (*Vehicle*1, *Car*1). *Vehicle*1 is a prototype of *Vehicle*, and *Car*1 is a prototype of *Car*, specified by "prototype" slots. *Vehicle* is a superclass of *Car*. KM has a rule that when a class inherits from its superclasses, all of the classes' instances will be cloned from the superclasses' prototypes. This means *Car*1 will be cloned from *Vehicle*1.

Therefore *Car*1 will obtain from *Vehicle*1 the engine's information. However, *Car*1 already has an engine; if it directly copies another engine from *Vehicle*1, then *Car*1 will have two engines, which is not intuitively correct. The better way involves *Car*1 keeping its engine, but that engine copies other attributes(values) from *Vehicle*1's engine, such as the attribute *powerful* as shown below.

```
(Car1 has
   (has-engine (a Engine with
      (strength (*Powerful))
      (size (*Average))
      (fuel (?)))
```

Again, both *Car*1's engine and *Vehicle*1's engine has the slot *fuel*. We again let *Car*1's *fuel* obtain attributes from *Vehicle*1's engine's fuel as follows:

```
(Car1 has
   (has-engine (a Engine with
      (strength (*Powerful))
      (size (*Average))
      (fuel (((a Gas-type with
            (type (*Unleaded))
```

```
(combustibility (*Hi)))))))
```

This whole process is called “cloning” and “unification,” which is an important aspect of reasoning in KM. KM defined it in a procedural way. In this work, we defined “clone” and “unification” in a declarative way.

3.3 Formal Definitions About a Frame Based Knowledge Base

In this section, we first formally define various aspects of a frame-based KB as the background for defining inheritance, cloning, and unification in the next section. Our following definitions are based on various frame-based systems such as KR, and we made some simplifications to focus on the main aspects: cloning and unification.

Frame-based KR focuses on representing two aspects: classes and instances. Class properties and hierarchy information are key aspects of each class. Class properties, which are inherited by all its instances, are represented in prototype in some frame-based KR formalisms like KR. The representation of prototypes is not different from the ones of normal instances, so we will start with the hierarchy information, defining the Class Hierarchy Graph.

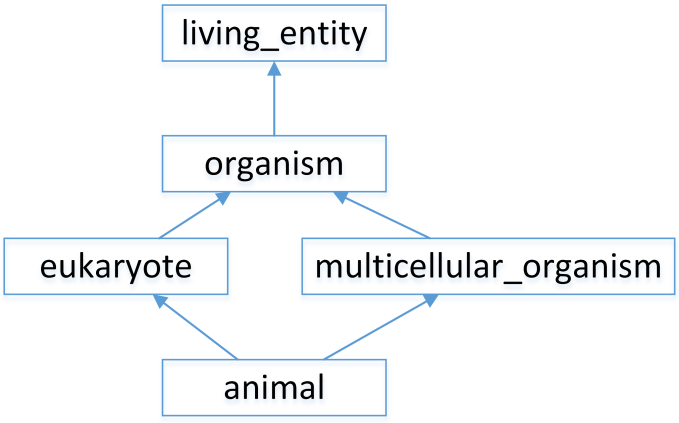

Figure 3.1: Snippet of Class Hierarchy Graph

Definition 17 *A Class Hierarchy Graph (G_{classH}) is a directed acyclic graph whose vertices are classes and edges encode class-superclass relationship; i.e. if (C1, C2) is a directed edge then C2 is a superclass of C1.*

The ASP encoding of G_{classH} is a collection of facts of the form:

```
has(X, superclass, Y).
class(X).
class(Y).
```

Where X and Y are vertices in the graph, and there is an edge from X to Y. Following is the encoding for the example in Figure 3.1:

```
has(animal, superclass, eukaryote).
has(animal, superclass, multicellular_organism).
has(eukaryote, superclass, organism).
has(multicellular_organism, superclass, organism).
has(organism, superclass, living_entity).
class(animal).
class(eukaryote).
class(multicellular_organism).
class(organism).
class(living_entity).
```

Definition 18 *An object graph $G_{obj}(O)$ (also denoted simply by $G(O)$) that describes an object O consists of a core object graph which is a directed graph that has following properties:*

1. *Its vertices are instances.*
2. *It has two types of edges: **non-inheritable** and **inheritable**.*
3. *All the vertices (or nodes) can be reached from O through a chain of inheritable edges and then an optional non-inheritable edge.*

*4. It is **acyclic** with respect to the **inheritable** edges.*

The ASP encoding of $G_{obj}(O)$ is a collection of all the slot values of X in the form $has(X,S,V)$, in which the slot S is the label of the edge connecting X to V.

An edge (X,R,Y) is labeled by R and means XRY. For example, an edge $(X, instance-of, Y)$ means that X is an instance of Y.

Non-inheritable labels, such as {*prototype-of*, *cloned-from*, etc}, tend to describe the "relations" rather than the "properties." These labels are not to be transferred from one object graph to another via cloning or inheritance.

Inheritable labels, such as {*instance-of*, *has-part*, *agent*, etc.}, describe the properties of an instance, and may be inherited or cloned by other instances.

In the following are the facts that we are going to use in many up-coming examples and illustrate how our definitions will work.

```
has(cell325, instance_of, cell).
has(cell325, prototype_participants, attach2542).
has(cell325, prototype_participants,
   negatively_charged_region11496).
has(cell325, prototype_participant_of, cell325).
has(cell325, prototype_of, cell).
has(cell325, prototype_scope, cell).
has(cell325, has_part, cytoplasm689).
has(cell325, has_part, ribosome23653).
has(cell325, has_part, chromosome10040).
has(cell325, has_part, enzyme6401).
has(cell325, has_part, plasma_membrane14508).
has(cell325, diameter, length_value15324).
has(cell325, has_region, cell_pole13862).
has(cell325, has_region, cell_pole13861).
has(cell325, has_region, surface1564).
```

```
has(cell325, cloned_from, cell18399).
has(cell325, cloned_from, tangible_entity22850).
has(cell325, clean_instance_of, cell).
```

Definition 19 *A Knowledge Base is a collection of object graphs, a class hierarchy graph, and a specification of inheritable/non-inheritable labels (slot names).*

Both object graphs and the class hierarchy graph are encoded using ASP facts of the form $has(X,S,V)$, in which X can be either an object or a class, S is a slot name, and V is the value for slot S. The noninheritable slots are encoded as $noninheritable(S)$.

Using the definitions above, we will define the transitive closure of "instance-of" and generalized version of "cloned-from" relations, which are useful for reasoning about a KB.

Definition 20 (tc-instance-of) *Let x be an instance and Y be a class in the knowledge base KB. x is a tc-instance-of Y iff one of the following conditions is satisfied:*

1. *x is an instance of Y*
2. *There exist a set of class $\{C_1, C_2, ..., C_k = Y\}$ satisfying all following conditions:*
   (a) *x is an instance of C_1*
   (b) *C_i is a subclass of C_{i+1} ($1 \leq i < k$)*

In the following example, $tc-instance-of$ relations $cell325$ and $cell$ (or $living\_entity$) are derived from *superclass* and $instance-of$ relation.

Example 12

```
has(cell, superclass, living_entity).
has(cell325, instance_of, cell).
  => tc_instance_of(cell325, cell).
     tc_instance_of(cell325, living_entity).
```

As a compact representation, $G_{obj}(X)$ (also denoted as $G(X)$) only contains instance-of edges connecting X to the immediate class(es) of X. However, X also recursively belongs to all the superclasses of the immediate class(es). The above definition defines a transitive-closure of the classes that an object X belongs to.

Using *tc-instance-of*, we define *g-cloned-from* relation which is a generalized version of the *cloned-from* relation. x is *g-cloned-from* y in two cases: (1) it is explicitly said that x is *cloned-from* y (see Figure 3.2(a)) and (2) x is a *tc-instance-of* class C, whose prototype is y (see Figure 3.2(b)). In the second case, by inheritance, x must have all properties of the prototype of C; thus, it must be cloned from y.

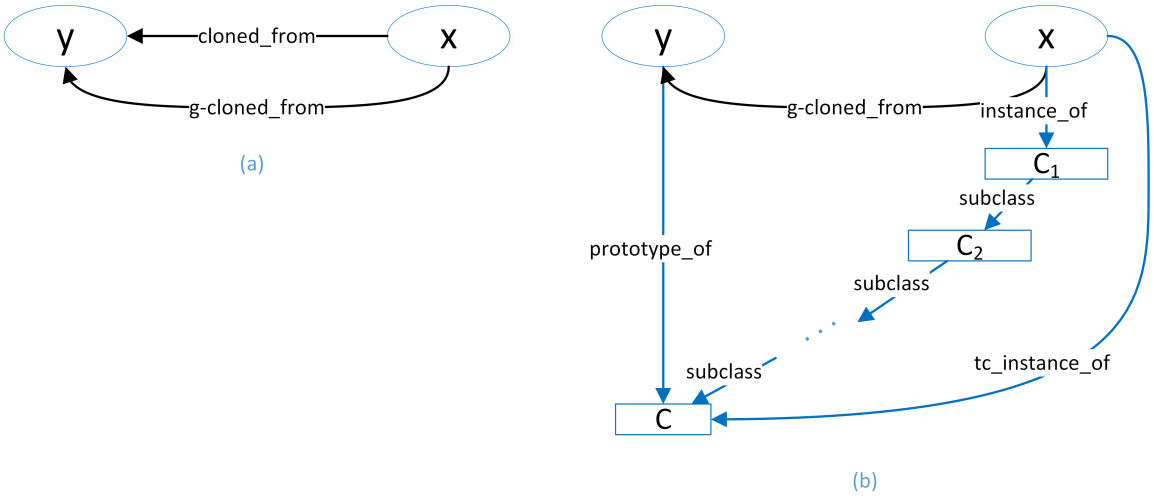

Figure 3.2: g-cloned-from relation

Definition 21 (g-cloned-from) *Let x and y be two instances in the knowledge base KB. x is g-cloned-from y iff*

- *x is cloned from y, or*
- *x is an tc-instance of a class c and y is a prototype of c*

In the following is an example of applying Definition (**?**).

Example 13

```
has(cell325, cloned_from, tangible_entity22850).
has(tangible_entity22850, instance, cell).
has(cell18399, prototype_of, cell).
  => g_cloned_from(cell325, tangible_entity22850).
     g_cloned_from(tangible_entity22850, cell18399).
```

To simplify the formulas and descriptions in the latter sections, given an edge label S, we will define the set of **nodes** connected from x or A through edges of label S.

Definition 22 ($N_P(x,S)$ and $N_P(A,S)$) *Let x be a node and A be a set of nodes in a set of object graphs P, and let S be an edge label.*
$N_P(x,S)$ *is the set of nodes* $\{n : (x,S,n)$ *in P* $\}$*, the set of nodes that link from x through an edge of type S in P.*
$N_P(A,S)$ *is union of* $N_P(x,S)$ *over all node x in A.*

For example, $N_{\{G(b1)\}}(b1, has-part)$ contains only one element: $body2$; $N_{\{G(a1)\}}(\{a1\}, has-part)$ contains two elements: $head1$ and $body1$.

3.4 Unification in a Frame-based Knowledge Base

$G(X)$ does not necessary contain complete information about X because extra information is in the object graphs containing $\{Y_1,...,Y_n\}$ which X is g-cloned-from. However, the properties of $\{Y_1,...,Y_n\}$ should not be directly added to X's object graph, but rather need to go through a merging process which is referred to as unification (Clark and Porter, 2011; Chaudhri and Son, 2012; Baral and Liang, 2012). In this section, we will describe the unification process and the principles that it follows.

3.4.1 Principles of Unification Process

Our unification process follows these two principles:

1. Respect the original annotation: The unification process on $G(X)$ should keep all the existing information of $G(X)$ in the KB.

2. Specificity principle: while acquiring extra information from $\{Y_1, ..., Y_n\}$, we keep only the most specific information. For example, if (x, S, a) and (x, S, b) are two possible candidate edges, we will keep the first one if a is more specific than b.

3.4.2 Specificness - Subsume Relation

Given two instances x and y, we need to define a measurement about the specificness. We define the measurement as subsume relation. x subsumes y, meaning x is more specific than y. The subsume relation is based on the classes that x and y are tc-instances-of.

Definition 23 (Subsume relation) *Let x and y be instances, the sets* $\{xc_1...xc_m\}$ *and* $\{yc_1...yc_n\}$ *be all the classes that x and y are tc-instances-of, respectively. We say that x subsumes y iff* $\{yc_1...yc_n\} \subseteq \{xc_1...xc_m\}$.

Since subsume relation is based on the subset relation, subsume relation is transitive: if x subsumes y and y subsumes z then x subsumes z.

In the following is an example of applying Definition 23.

Example 14 (Subsume)

```
tc_instance_of(cell325, cell).
tc_instance_of(cell325, living_entity).
tc_instance_of(tangible_entity22850, cell).
tc_instance_of(tangible_entity22850, tangible_entity).
has(cell, superclass, living_entity).
has(cell, superclass, tangible_entity).
=>    tc_instance_of(cell325, tangible_entity).
      tc_instance_of(tangible_entity22850, living_entity).
```

```
=>     subsume(cell325, tangible_entity22850)
       subsume(tangible_entity22850, cell325)
```

In this case, since *cell* is a subclass of *tangible_entity* and of *living_entity*, *cell*325 is also a tc-instance-of *tangible_entity* and *tangible_entity*22850 is also a tc-instance-of *living_entity*. *cell*325 and *tangible_entity*22850 therefore are both tc-instance-of 3 classes: *cell*, *living_entity*, and *tangible_entity*. Thus, *cell*325 and *tangible_entity*22850 subsume each other.

Since the subsume relation is based on the subset relation of sets, it is a partial order. Given a set of nodes, we can define the most specific or most detailed nodes as the minimal elements according to subsume relation.

Definition 24 (Minimal with respect to subsume relation) *Let A be a set of nodes in a knowledge base KB. Let* $MIN_{subsume}(A)$ *be a set of minimal elements of A according to subsume relation. When two nodes in A subsume each other,* $MIN_{subsume}(A)$ *contains a random one.*

For example, Figure 3.3 shows the subsume relation among all the instances in the set $A = \{x_0, x_1, ..., x_5\}$. x_0 and x_1 are clearly two minimal elements. Since x_4 and x_5 subsume each others, either of them will be in $MIN_{subsume}(A)$. The two possible $MIN_{subsume}(A)$ therefore are $\{x_0, x_1, x_4\}$ and $\{x_0, x_1, x_5\}$.

Given two sets A and B, suppose we want to keep all the nodes in B and add some extra nodes in A to it. To do that, we would limit checking only the set of more specific nodes in A. In another words, remove from A all the nodes in A which are less specific than any node in B. We define the result set as the difference between A and B with respect to subsume relation.

Definition 25 (Set difference with respect to subsume relation) *Let A and B be two set*

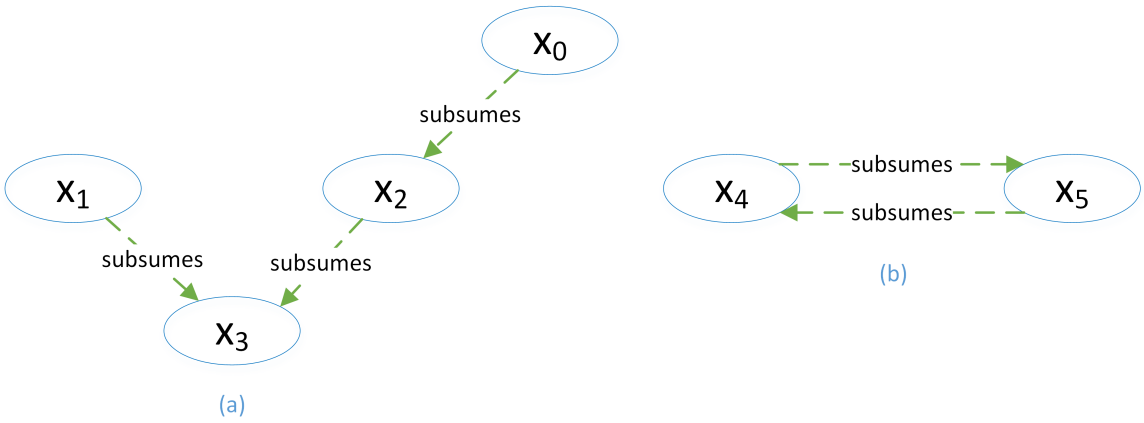

Figure 3.3: Subsume relationall the instances in the set $A = \{x_0, x_1, ..., x_5\}$. The two possible $MIN_{subsume}(A)$ thus are $\{x_0, x_1, x_4\}$ and $\{x_0, x_1, x_5\}$

of nodes in a knowledge base KB. Let $A \setminus_{subsumed} B$ *denote the set of A's elements which is not subsumed by any of B's element.*

Since x subsumes itself, $A \setminus_{subsumed} B$ does not contains any node of B: $(A \setminus_{subsumed} B) \cap B = \emptyset$

3.4.3 Microclone and Unification Process

Let us consider the following example shown in Figure 3.4 .

- *mammal* is subclass of *animal*.
- $a1$ is a prototype (and of course also an instance) of *animal*.
- $b1$ is a prototype of *mammal*.
- $d1$ is a prototype of $mutated\_animal$.
- $c1$ is an instance of $mutated\_dog$. $c1$ is cloned from $b1$.
- $a1$ has $body1$ and $head1$. $body1$ has $big1$.

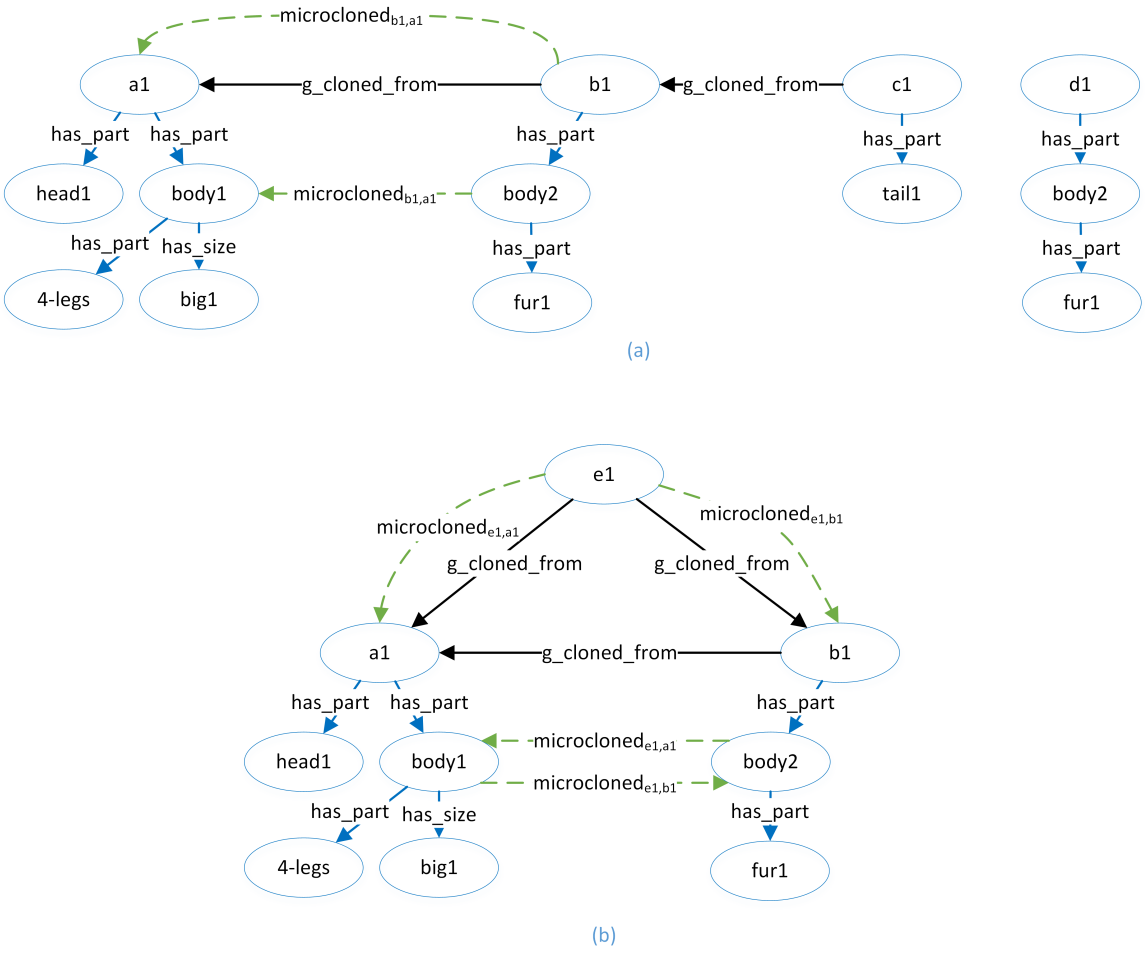

Figure 3.4: (a) Object graphs for $a1$, $b1$, $c1$, and $d1$ together with their relations (b) Object graphs for $a1$, $b1$, and $e1$ together with their relations

- $b1$ has $body2$ and $4-legs$. $body2$ is cloned from $body0$.
- $c1$ has $tail1$.
- $d1$ also has $body2$.
- $e1$ is an instance of *animal* and it is cloned from both $a1$ and $b1$.

Example 15 (ASP encoding of the example in Figure 3.4)

```
has(a1, instance_of, animal).
```

```
has(a1, prototype_of, animal).
has(a1, has_part, body1).
has(a1, has_part, head1).
has(body1, instance_of, body).
has(body1, has_size, big1).
has(body1, has_part, four_legs).
has(four_legs, instance_of, legs).
has(head1, instance_of, head).
has(big1, instance_of, size).

has(b1, instance_of, mammal).
has(b1, prototype_of, mammal).
has(b1, has_part, body2).
has(body2, instance_of, body).
has(body2, has_part, fur1).
has(fur1, instance_of, fur).

%%%
has(body2, cloned_from, body0).
has(body0, instance_of, body).
%%%

has(c1, instance_of, mutated_dog).
has(c1, cloned_from, b1).
has(c1, has_part, tail1).
has(tail1, instance_of, tail).

has(d1, instance_of, mutated_animal).
has(d1, prototype_of, mutated_animal).
has(d1, has_part, body2).
```

```
has(e1, instance_of, animal).
has(e1, cloned_from, a1).
has(e1, cloned_from, b1).

% Class information
class(animal).
class(mammal).
class(mutated_dog).
class(mutated_animal).

class(tail).
class(body).
class(size).
class(head).
class(legs).
class(fur).

has(mammal, superclass, animal).
```

Since $b1$ is an instance of *mammal*, it should inherit all properties of the prototype $a1$ of *animal* (or $b1$ should be cloned from $a1$.) By stating this, it should be understood that $b1$ should have $4 - legs$ from $body1$ of $a1$. In other words, $body2$ should be cloned from $body1$. However, since $d1$ is not cloned from $a1$, it should not have $4 - legs$, even though it has $body2$. In order to differentiate these cases, we define the context and microclone relation in such a way that $body2$ is only cloned from $body1$ in the context of $b1$, and that cloning relation is named microclone.

More specifically, $b1$ should be microcloned from $a1$ since it is defined to be cloned from $a1$; $body2$ should be microcloned from $body1$ because $b1$ is cloned from $a1$ and they play the same role in $b1$ and $a1$ (they both are the bodies of $a1$ and $b1$ respectively).

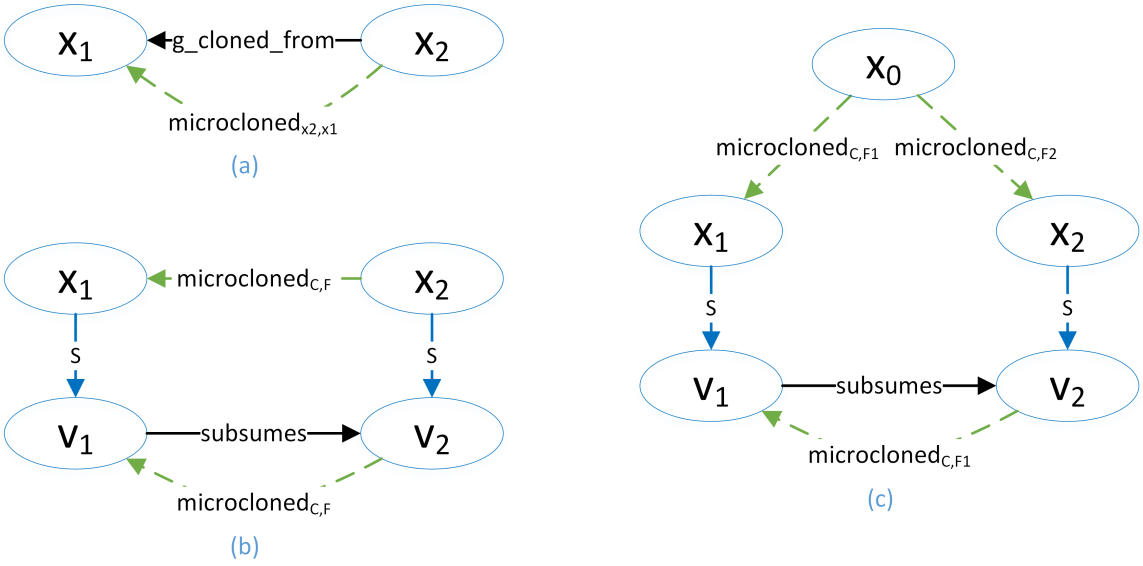

Figure 3.5: Three cases of microcloning. The arrow labeled $microcloned_{b1,a1}$ from $body2$ to $body1$ means that $body2$ in $b1$ is microcloned from $body1$ in $a1$.

Figure 3.5 illustrates three cases of microcloning in definition 26. The first case in 3.5(a) is straight-forward, when x_2 is g-cloned from x_1, x_2 in the context of x_2 (or "x_2 in x_2" for short) is microcloned from x_1 in x_1. In another words, we can also understand that x_2 in $G(x_2)$ is microcloned from x_1 in $G(x_1)$.

The intuition behind Figure 3.5(b) is that when we unify $G(x_2)$ with $G(x_1)$, v_1 contains more specific information than v_2 but we have to keep the name v_2, according to the first principle. In order to preserve the information, v_2 should be marked to be microcloned from v_1 so that later, $G(v_2)$ will be unified with $G(v_1)$ to get extra information.

In the case of Figure 3.5(c), when both v_1 and v_2 should be considered for adding to x_0, v_2 should be microcloned from v_1 so if v_2 is chosen to be added to x_0, the details in v_1 will be considered while unifying v_1. In the latter of this section, this will not happen in cases where v_1 is more specific than v_2 and we choose v_1 to add to x_0. However, when v_1 and v_2 are equally specific (v_1 and v_2 subsume each other like $body1$ from $a1$ and $body2$ from

$b2$ in Figure 3.4(b), they will be marked to be microcloned from each other. Randomly selecting one of them will not lose us any information.

The formal definition of microcloned is given in the following.

Definition 26 *Let x_1, x_2, v_1, v_2, C and F be instances in a knowledge base KB.*

1. *x_2 in the context of x_2 (or in x_2 for short) is microcloned from x_1 in x_1 if x_2 is g-cloned x_1. (see Figure 3.5(a))*

2. *v_2 in C is microcloned from v_1 in F if all of the following conditions are satisfied (see Figure 3.5(b))*

   (a) *x_2 in C is microcloned from x_1 in F*

   (b) *KB contains two inheritable edges (x_1, S, v_1) and (x_2, S, v_2)*

   (c) *v_1 subsumes v_2*

3. *v_2 is microcloned from v_1 in C if all of the following conditions are satisfied (see Figure 3.5(c))*

   (a) *x_1 in C is microcloned from x_0 in F*

   (b) *x_2 in C is microcloned from x_0 in F*

   (c) *KB contains two inheritable edges (x_1, S, v_1) and (x_2, S, v_2)*

   (d) *v_1 subsumes v_2*

When the context C is clear, we only use "x_2 is microcloned from x_1".

In the example in Figure 3.4, $b1$ in $b1$ is microcloned from $a1$ in $a1$ because $b1$ is tc-cloned-from $a1$. $body2$ in $b1$ is microcloned from $body1$ in $a1$ according to definition 26.2: $body1$ subsumes $body2$ (they actually subsume each other), KB contains

$(a1, has\_part, body1)$ and $(b1, has\_part, body2)$, and $b1$ in $b1$ is microcloned from $a1$ in $a1$.

Using Definition 26, we can have two useful lemma about a node microcloned from any node. If a node x in $G(C)$ cannot be reached from C through inheritable edges only (in other words, it must travel through non-inheritable edge to reach x), x is not microcloned from any node in the context of C.

Lemma 9 *Let v, v′, C, and F be instances in a knowledge base KB. If v in C is microcloned from v′ in F (v ≠ C), there exists a path of inheritable edges from C to v.*

Proof 9

1. *Because v ≠ C, v in C is microcloned from v′ in F comes from the last two cases of Definition 26. This means that the parent x of v is microcloned from x′ in F where x connects to v through an inheritable edge.*
2. *x can be C or not. If x ≠ C, the parent node x_2 of x is also microcloned from another node; x_2 connects to x through an inheritable edge. ...*
3. *The process is repeated until $x_k = C$ is achieved. We can go from $x_k = C$ through x_{k-1}, x_{k-2}, ..., x_2, x and to v by inheritable edges.*

Lemma 10 *Let v is a node in $G(C)$ in a knowledge base KB. If the last edge of all the paths from C to v is non-inheritable, v is not microcloned from any node in the context of C.*

Proof 10 *This can be proved through contradiction using previous lemma. If v is microcloned from a node in the context of C, there must be a path of inheritable edges from C to v.*

Armed with the subsume and microcloned relation, we define the unification process as the following.

First of all, given an object graph $G(C)$ and a set of object graphs P that we need to unify to $G(C)$, we retain all $G(C)$'s nodes and edges in the unification result. For each node x of $G(C)$, it may be microcloned from some nodes x_i in P. These nodes x_i may contain additional information that x should have. This information is contained in the edges (x_i, S, v_j) and $G(v_j)$ in P. We will merge this information into $G(C)$ by creating the extended set of edges (x, v_j) and by including the set as well as $G(v_j)$ in the unification result.

While this approach obviously satisfies the first principle of unification aforementioned, the extended set of edges (x, v_j) must be chosen according to the second principle: v_js must be the most specific nodes. The process of selecting the most specific nodes must be done according to each possible type of edge S. Let M be the set of nodes in $G(C)$ reached from x through edge S; let N be the ones in P from x_is. Since M will be in the unification result, we would want to remove all nodes in N that have a more specific one in M. After that, the most specific nodes will be determined based on "subsume" relation. Thus, the extended set of edges will contains the edges (x, v_j) where v_js are in $MIN_{subsume}(N \setminus_{subsume} M)$.

Following is the formal definition of the extended set of edges.

Definition 27 (Extended set of edges) *Let x and C be two instances in a knowledge base KB, P be a set of object graphs. Let A be the set of instances in P that x in the context C is microcloned from.*

An extended set of edges of label S from x in the context C, denoted by $edges^*_C(x, S, P)$ *is set of edges from x to all instances in the set of nodes* $MIN_{subsume}(N_P(A, S) \setminus_{subsume} N_{G(C)}(x, S))$.

An extended set of edges from x in the C, denoted by $edges^*_C(x, P)$ *is the union of* $edges^*_C(x, S, P)$ *over all labels S.*

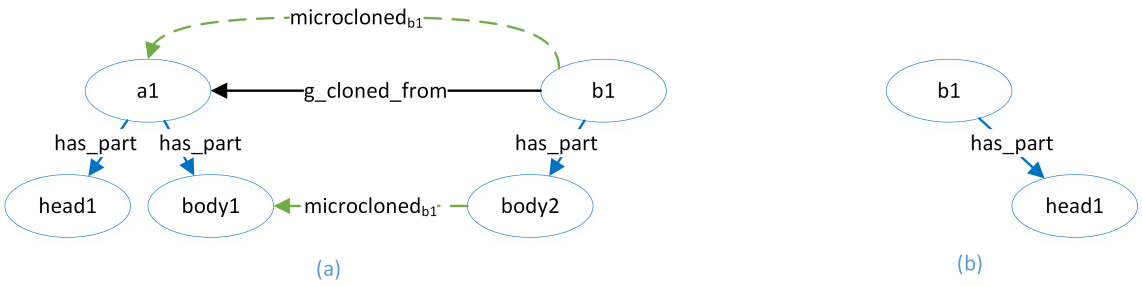

Figure 3.6: Enhancement set of edges from $b1$ in $b1$ in unification $G(b1)$ with $G(a1)$

(a): relations between $a1$ and $b1$

(b): Enhancement set of edges from $b1$ in $b1$

In the definition, $N_{G(C)}(x,S)$ is the set of nodes in $G(C)$ reached from x through edges of type S, $N_P(A,S)$ is the set of nodes in P reached from any node in A through edges of type S.

The $\setminus_{subsume}$ between two sets is to remove the less specific nodes. From this different set, we get only the most specific elements by $MIN_{subsume}()$.

If x is not microcloned from any instance ($A = \emptyset$), obviously the extended set of edges is empty.

Following is an example how an extended set of edges from $b1$ in the context of $b1$ is obtained.

Example 16 (Obtaining a $edges^*_{b1}(b1)$, the extended set of edges from $b1$ in the context of $b1$) *As shown before, $b1$ in the context $b1$ ("$b1$ in $b1$" for short) is microcloned from only $a1$ in $a1$. The set of all instances that $b1$ in $b1$ is microcloned from is $A = \{a1\}$.*

We start by computing the extended set of edge type $has-part$.

1. *Calculate set of nodes from $b1$ and A through edges $has-part$: $N_{G(b1)}(b1, has-part) = \{body2\}$ and $N_{G(a1)}(\{a1\}, has-part) = \{head1, body1\}$.*

2. *Calculate $N_P(A,S) \setminus_{subsume} N_{G(C)}(x,S)$, the addition nodes that we should link to $b1$.*

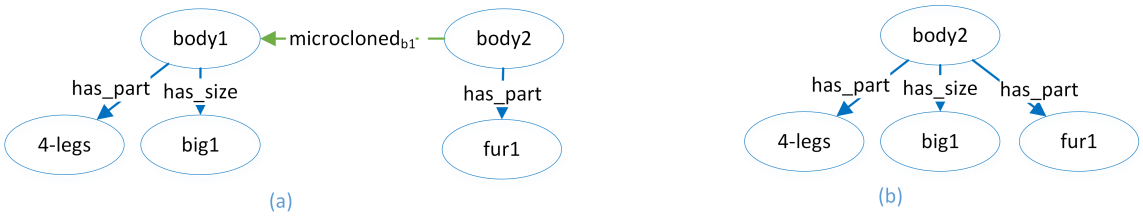

Figure 3.7: Extended set of edges from $body2$ with respect to $b1$ in unification $G(b1)$ with $G(a1)$ (a): relations between $body1$ and $body2$
(b): Extended set of edges from $body2$ with respect to $b1$

Since $body1$ *of* $a1$ *is subsumed by* $body2$ *of* $b1$*, it is excluded.* $N_P(A,S) \setminus_{subsume} N_{G(C)}(x,S)$ *contains only* $head1$.

3. *Find the most specific nodes in* $N_P(A,S) \setminus_{subsume} N_{G(C)}(x,S)$. $head1$ *is the only node and the most specific one in this set:* $MIN_{subsume}(\{head1\}) = \{head1\}$

4. *Add a new* $has-part$ *edge from* $b1$ *to* $body1$ *into the extended set.*

The extended set of edges of type $has-part$ *from* $b1$, $edges^*_C(b1, has-part)$, *thus has only one edge.*

Since $has-part$ *is the only type of edges from* $b1$ *and A,* $edges^*_C(b1)$ *is also* $edges^*_C(b1, has-part)$ *and contains only one edge (Figure 3.6).*

As explained earlier, in order to unify $G(b1)$ with $G(a1)$, we need to compute the extended set of edges from each node of $b1$. For example, figure 3.6 shows the extended set of edges from $b1$ in $b1$; figure 3.7 shows the extended set of edges from $body2$ in $b1$. Since $body0$ is not microcloned from any node, the extended set of edges from it is empty. Combine $G(b1)$ with all the extended sets of edges and we have the unification of $G(b1)$ and $G(a1)$ as shown in Figure 3.8. If $G(head1)$, $G(4-legs)$, or $G(big1)$ contains more nodes, we would have to include them in the unification result.

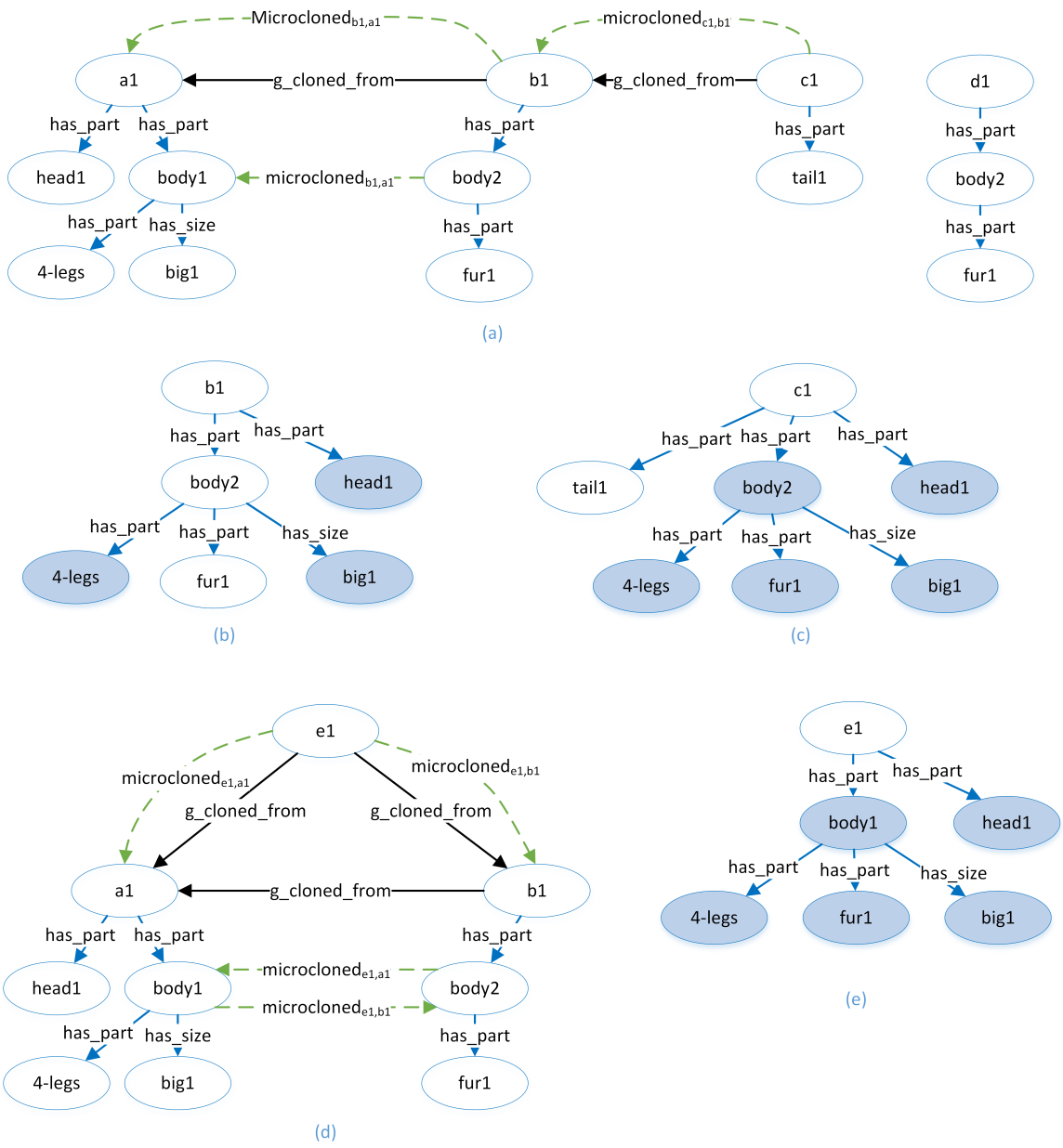

Figure 3.8: Unified enhancement graphs. Shaded nodes are added by the unification process.

(a) Object graphs for $a1$, $b1$, $c1$, and $d1$ together with their relations

(b) Unified enhancement of $G(b1)$. d (c) Unified enhancement of $G(c1)$

(d) Object graphs for $a1$, $b1$, and $e1$ together with their relations

(e) Unified enhancement of $G(e1)$

The formal definition of the unification of an object graph is given as follows.

Definition 28 (Unified enhancement of an object graph in a knowledge base) *The unified enhancement of object graph $G(x)$ of a knowledge base KB, denoted as $G^*(x)$.*

1. *When x is not g-cloned from any node in KB, $G^*(x)$ is $G(x)$ itself.*
2. *When x is g-cloned from x_1, x_2,... x_k, $G^*(x)$ is obtained from $G(x)$ by adding to each node v of $G(x)$ the following:*
   (a) *Each edges (v,t_j) in edges $*_x$ $(v, \{G^*(x_1), G^*(x_2), ..., G^*(x_k)\})$, where t_j is obtained from $G^*(x_i)$, and*
   (b) *all the graphs $G(t_j)$ in $G^*(x_i)$*

Below, we show how the unification process works for the example in Figure 3.4.

Example 17 ($G^*(a1)$, unified enhancement of object graph $G(a1)$) *Since $a1$ is not cloned from any other nodes, $G^*(a1)$ is $G(a1)$.*

Example 18 ($G^*(b1)$, unified enhancement of object graph $G(b1)$) *Since $b1$ is g-cloned from $a1$, the unified enhancement of $G(b1)$ is based on $G^*(a_1)$. As shown above, $G^*(b1)$ is obtained by unifying $G(b1)$ and $G(a1)$ and is shown in Figure 3.8. After unification, there are some more nodes merged to $G(b1)$ (shaded nodes in Figure 3.8(b)).*

Example 19 ($G^*(c1)$, unified enhancement of object graph $G(c1)$) *The unified enhancement of object graph $G(c1)$ is calculated based on $G^*(b1)$. Start by calculating the extended set of edges from $c1$; since $c1$ is microcloned from $b1$, we have to consider nodes $body2$ and $head1$ from $b1$. Because they are all the most specific nodes, the extended set of edges from $c1$ contains two edges from $c1$ to $body2$ and to $head1$. There is no other node in $G(c1)$ microcloned from other nodes in $G(b1)$. The $G^*(c1)$ is constructed by combining together $G(c1)$, two edges from $c1$ to $body2$ and to $head1$, $G(body2)$, and $G(head1)$.*

Compared to $G(c1)$, $G^*(c1)$ has some more nodes obtained from $G(body2)$ and $G(head1)$ (shaded nodes in Figure 3.8(c))

Example 20 ($G^*(d1)$, unified enhancement of object graph $G(d1)$) *Since $d1$ is not cloned from any other nodes, $G^*(d1)$ is $G(d1)$.*

Example 21 ($G^*(e1)$, unified enhancement of object graph $G(e1)$) *Either $head1$ in $b1$ or $head1$ in $a1$ is put into $G^*(e1)$, because they subsume each other. Note that although these two are both $head1$, their children may differ. Similarly, either $body1$ in $a1$ or $body2$ in $b1$ is selected. However, in either case, since they subsume each other, the children of the other node are also put into $G^*(e1)$. $G^*(e1)$ contains the same amount of information as $G^*(b1)$.*

3.5 Answer Set Programming Encoding of Frame-based Reasoning Aspects

In this section we present the declarative ASP rules for reasoning over the knowledge base.

From the previous section we noticed several properties of the knowledge base. In particular, the representation of an object is not self-contained. An object graph describing an object X only records the most specific properties of X, and it will let X inherit from the superclasses or clone from other instances to obtain the general properties. One of the fundamental reasoning tasks for the KB is to retrieve the full properties for an object, which can be used to answer the most fundamental question: **What is X?**. The **property acquiring** process for an instance has the following steps:

- **Encoding basic information of the Knowledge Base:** encoding the various components of a Knowledge Base.
- **Obtaining generalizations of the instance and subsume relation:** Each instance can have facts about inheritance in the form: $has(Instance1, instance\_of, Class1)$,

and each class can have facts expressing superclass information:
$has(Class1, superclass, Class2)$, which indicates that $Class2$ is a superclass of $Class1$. Therefore, $Instance1$ will inherit from $Class2$ as well. Transitive closure rules are needed to define this process.

- **Obtaining what an instance clones from:** Besides inheritance, an instance can also obtain some properties from other instances via cloning:
$has(Instance1, clone\_from, Instance2)$. It can also obtain from the prototype of the classes it belongs to.

- **The unification process:** When inheriting or cloning happens, an instance is required to acquire information from the other classes/instances. Each slot value of the instance will not just be the one stated in its own frame, but will be merged with all the acquired information. The merging process is referred to as unification.

We will continue using the example in Figure 3.4 to illustrate the unification process step by step. The input ASP encoding is as follows:

```
has(a1, instance_of, animal).
has(a1, prototype_of, animal).
has(a1, has_part, body1).
has(a1, has_part, head1).
has(body1, instance_of, body).
has(body1, has_size, big1).
has(body1, has_part, four_legs).
has(four_legs, instance_of, legs).
has(head1, instance_of, head).
has(big1, instance_of, size).

has(b1, instance_of, mammal).
has(b1, prototype_of, mammal).
```

```
has(b1, has_part, body2).
has(body2, instance_of, body).
has(body2, has_part, fur1).
has(fur1, instance_of, fur).

%%%
has(body2, cloned_from, body0).
has(body0, instance_of, body).
%%%

has(c1, instance_of, mutated_dog).
has(c1, cloned_from, b1).
has(c1, has_part, tail1).
has(tail1, instance_of, tail).

has(d1, instance_of, mutated_animal).
has(d1, prototype_of, mutated_animal).
has(d1, has_part, body2).

has(e1, instance_of, animal).
has(e1, cloned_from, a1).
has(e1, cloned_from, b1).

% Class information
class(animal).
class(mammal).
class(mutated_dog).
class(mutated_animal).

class(tail).
class(body).
```

```
class(size).
class(head).
class(legs).
class(fur).

has(mammal, superclass, animal).
```

3.5.1 Encoding Basic Information of the Knowledge Base

Encoding the Specification of Non-Inheritable/inheritable Slot Names

Object graphs and the Hierarchy graph of the KB are specified above. The specification of non-inheritable/inheritable slot names (edge labels) are specified by "noninheritable(S)" predicate such as:

a1:

```
noninheritable(prototype_participants;
      prototype_participant_of;
      prototype_scope;
      prototype_of;
      cloned_from;
      clone_built_from;
      instance_of;
      clean_instance_of;
      big_nodes).
```

We do not need another predicate for inheritable slot names because all slot names that are not non-inheritable are considered inheritable.

Encoding Instances/Classes

All the classes in the KB are encoded by predicate "class(C)". All other atoms appeared in "has(X, _, V)" are considered as instances and are encoded by predicate "ins( )".

a2, a3:

```
% grab all instances
ins(X) :- has(X,S,V), not class(X).
ins(V) :- has(X,S,V), not class(V).
```

Following are instances and classes in the example input. As expected, all the instances are recognized correctly in the output of running those rules on previous example facts:

```
class(animal)
class(body)
class(fur)
class(head)
class(legs)
class(mammal)
class(mutated_animal)
class(mutated_dog)
class(size)
class(tail)
ins(a1)
ins(b1)
ins(big1)
ins(body0)
ins(body1)
ins(body2)
ins(c1)
ins(d1)
ins(e1)
ins(four_legs)
ins(fur1)
ins(head1)
ins(tail1)
```

3.5.2 Obtaining Generalizations of an Instance

When retrieving information for an instance, the first step is to gather all of its classes. Here, we discuss the rules for obtaining multiple inheriting classes for an instance. The slot "instance-of" directly encodes all the classes (immediate class) to which an instance belongs to, and the instance also recursively belongs to the immediate classes' superclasses.

The rules $a4$ and $a5$ encodes $instance\_of(X,Y)$, which means class Y is an immediate class of instance X.

a4, a5:

```
instance_of(X,Y) :- has(X,instance_of,Y).
instance_of(X,Y) :- has(X,clean_instance_of,Y).
```

The rules $a6$ and $a7$ encode $g\_instance\_of(X,Y)$, which means instance X transitively belongs to class Y. Rule $i3$ uses the immediate class as the base case. Rule $a7$ means that if X is a $tc\_instance\_of\ M$ and Y is a superclass of M, then X is also a $tc\_instance\_of\ Y$.

a6, a7:

```
tc_instance_of(X,Y) :- instance_of(X,Y).
tc_instance_of(X,Y) :- tc_instance_of(X,M), has(M,superclass,Y).
```

Shown in the following is output of running $tc\_instance\_of$ rules:

```
% Here is the running output of tc_instance_of:
      tc_instance_of(a1,animal)
      tc_instance_of(b1,animal)
      tc_instance_of(b1,mammal)
      tc_instance_of(big1,size)
      tc_instance_of(body0,body)
      tc_instance_of(body1,body)
      tc_instance_of(body2,body)
      tc_instance_of(c1,mutated_dog)
```

```
tc_instance_of(d1,mutated_animal)
tc_instance_of(e1,animal)
tc_instance_of(four_legs,legs)
tc_instance_of(fur1,fur)
tc_instance_of(head1,head)
tc_instance_of(tail1,tail)
```

As expected, the instance of subclass, like $b1$ of *mammal*, is also tc-instance-of super-class, like *animal*.

3.5.3 Encoding Subsume Relation

The subsume relation is encoded by rules $a8$ and $a9$. $V1$ does not subsume $V2$ if there exists a class C which $V2$ is a tc-instance-of but $V1$ is not. $V1$ subsumes $V2$ if there is no evidence telling otherwise.

a8, a9:

```
% V1 subsumes V2 if V1 is more specific.
not_subsume(V1,V2) :-
      ins(V1), ins(V2),
      tc_instance_of(V2,C),
      not tc_instance_of(V1,C),
      V1!=V2.

subsume(V1,V2) :-
      ins(V1), ins(V2),
      not not_subsume(V1,V2).
```

In the following are the output facts from running the two rules above on our example input of $a1$, $b1$, $c1$, and $d1$. As we can see, $a1$, $b1$, $c1$, and $d1$ subsume each other because the only class they are instance of is *animal*. $body0$, $body1$, and $body2$ subsume each other, since the only class they belong to is *body*. The relations between other pairs are all

not_subsume.

```
not_subsume(a1,b1)
not_subsume(a1,big1)
not_subsume(a1,body0)
not_subsume(a1,body1)
not_subsume(a1,body2)
not_subsume(a1,c1)
not_subsume(a1,d1)
not_subsume(a1,four_legs)
not_subsume(a1,fur1)
not_subsume(a1,head1)
not_subsume(a1,tail1)
not_subsume(b1,big1)
not_subsume(b1,body0)
not_subsume(b1,body1)
not_subsume(b1,body2)
not_subsume(b1,c1)
not_subsume(b1,d1)
not_subsume(b1,four_legs)
not_subsume(b1,fur1)
not_subsume(b1,head1)
not_subsume(b1,tail1)
not_subsume(big1,a1)
not_subsume(big1,b1)
not_subsume(big1,body0)
not_subsume(big1,body1)
not_subsume(big1,body2)
not_subsume(big1,c1)
not_subsume(big1,d1)
not_subsume(big1,e1)
not_subsume(big1,four_legs)
```

```
not_subsume(big1,fur1)
not_subsume(big1,head1)
not_subsume(big1,tail1)
not_subsume(body0,a1)
not_subsume(body0,b1)
not_subsume(body0,big1)
not_subsume(body0,c1)
not_subsume(body0,d1)
not_subsume(body0,e1)
not_subsume(body0,four_legs)
not_subsume(body0,fur1)
not_subsume(body0,head1)
not_subsume(body0,tail1)
not_subsume(body1,a1)
not_subsume(body1,b1)
not_subsume(body1,big1)
not_subsume(body1,c1)
not_subsume(body1,d1)
not_subsume(body1,e1)
not_subsume(body1,four_legs)
not_subsume(body1,fur1)
not_subsume(body1,head1)
not_subsume(body1,tail1)
not_subsume(body2,a1)
not_subsume(body2,b1)
not_subsume(body2,big1)
not_subsume(body2,c1)
not_subsume(body2,d1)
not_subsume(body2,e1)
not_subsume(body2,four_legs)
not_subsume(body2,fur1)
```

```
not_subsume(body2,head1)
not_subsume(body2,tail1)
not_subsume(c1,a1)
not_subsume(c1,b1)
not_subsume(c1,big1)
not_subsume(c1,body0)
not_subsume(c1,body1)
not_subsume(c1,body2)
not_subsume(c1,d1)
not_subsume(c1,e1)
not_subsume(c1,four_legs)
not_subsume(c1,fur1)
not_subsume(c1,head1)
not_subsume(c1,tail1)
not_subsume(d1,a1)
not_subsume(d1,b1)
not_subsume(d1,big1)
not_subsume(d1,body0)
not_subsume(d1,body1)
not_subsume(d1,body2)
not_subsume(d1,c1)
not_subsume(d1,e1)
not_subsume(d1,four_legs)
not_subsume(d1,fur1)
not_subsume(d1,head1)
not_subsume(d1,tail1)
not_subsume(e1,b1)
not_subsume(e1,big1)
not_subsume(e1,body0)
not_subsume(e1,body1)
not_subsume(e1,body2)
```

```
not_subsume(e1,c1)
not_subsume(e1,d1)
not_subsume(e1,four_legs)
not_subsume(e1,fur1)
not_subsume(e1,head1)
not_subsume(e1,tail1)
not_subsume(four_legs,a1)
not_subsume(four_legs,b1)
not_subsume(four_legs,big1)
not_subsume(four_legs,body0)
not_subsume(four_legs,body1)
not_subsume(four_legs,body2)
not_subsume(four_legs,c1)
not_subsume(four_legs,d1)
not_subsume(four_legs,e1)
not_subsume(four_legs,fur1)
not_subsume(four_legs,head1)
not_subsume(four_legs,tail1)
not_subsume(fur1,a1)
not_subsume(fur1,b1)
not_subsume(fur1,big1)
not_subsume(fur1,body0)
not_subsume(fur1,body1)
not_subsume(fur1,body2)
not_subsume(fur1,c1)
not_subsume(fur1,d1)
not_subsume(fur1,e1)
not_subsume(fur1,four_legs)
not_subsume(fur1,head1)
not_subsume(fur1,tail1)
not_subsume(head1,a1)
```

```
not_subsume(head1,b1)
not_subsume(head1,big1)
not_subsume(head1,body0)
not_subsume(head1,body1)
not_subsume(head1,body2)
not_subsume(head1,c1)
not_subsume(head1,d1)
not_subsume(head1,e1)
not_subsume(head1,four_legs)
not_subsume(head1,fur1)
not_subsume(head1,tail1)
not_subsume(tail1,a1)
not_subsume(tail1,b1)
not_subsume(tail1,big1)
not_subsume(tail1,body0)
not_subsume(tail1,body1)
not_subsume(tail1,body2)
not_subsume(tail1,c1)
not_subsume(tail1,d1)
not_subsume(tail1,e1)
not_subsume(tail1,four_legs)
not_subsume(tail1,fur1)
not_subsume(tail1,head1)
subsume(a1,a1)
subsume(a1,e1)
subsume(b1,a1)
subsume(b1,b1)
subsume(b1,e1)
subsume(big1,big1)
subsume(body0,body0)
subsume(body0,body1)
```

```
subsume(body0,body2)
subsume(body1,body0)
subsume(body1,body1)
subsume(body1,body2)
subsume(body2,body0)
subsume(body2,body1)
subsume(body2,body2)
subsume(c1,c1)
subsume(d1,d1)
subsume(e1,a1)
subsume(e1,e1)
subsume(four_legs,four_legs)
subsume(fur1,fur1)
subsume(head1,head1)
subsume(tail1,tail1))
```

As we can predict, because $b1$ is an instance of *mammal*, it subsumes both $a1$ and $e1$. $e1$ and $a1$ subsume each other because they are both instances of *animal*. $d1$ is the only instance of $mutated\_animal$ and does not subsume any other instance.

3.5.4 Obtaining What an Instance Clones From

Cloning is another mechanism that facilitates the reuse of the existing knowledge frames and avoids repeating the encoding of the same set of knowledge entries. **Default cloning** refers to the case where the information encoded for a prototype is transferred to other instances of the same class. **User-defined cloning** refers to the user specified facts of the form $has(X, cloned\_from, Y)$. Both cases need to be taken into account when deciding the set of instances the current instance is cloning from.

The rules $a10$ and $a11$ encode the **User-defined cloning**, rule $a12$ encodes the **Default cloning**.

a10, a11, a12:

```
g_cloned_from(X,Y) :- X!=Y, has(X,cloned_from,Y).
g_cloned_from(X,Y) :- X!=Y, has(X,clone_built_from,Y).
g_cloned_from(X,Y) :- X!=Y, tc_instance_of(X,M), has(Y,
   prototype_of,M).
```

In the following is the output related to g-cloned-from rules. This output matches the g-cloned-from relations illustrated in Figure 3.4.

```
g_cloned_from(b1,a1)
g_cloned_from(body2,body0)
g_cloned_from(c1,b1)
g_cloned_from(e1,a1)
g_cloned_from(e1,b1)
```

Encoding microclones relation Having the *g_cloned_from* relation, we can define the *microclone* relations. Rules $a13$, $a14$, and $a15$ respectively encode the three cases of microcloning in Definition 26.

a13, a14, a15:

```
% Case 1:
microclones((X1,X1), (X2,X2)) :-
      ins(X1), ins(X2),
      g_cloned_from(X1,X2).

% Case 2:
microclones((V1,C), (V2,F)) :-
      microclones((X1,C), (X2,F)),
      has(X1, S, V1),
      has(X2, S, V2),
      not noninheritable(S),
      subsume(V1,V2), V1!=V2.
```

```
% Case 3:
microclones((V1,F1), (V2,F2)) :-
      microclones((X,C), (X1,F1)),
      microclones((X,C), (X2,F2)),
      has(X1, S, V1),
      has(X2, S, V2),
      not noninheritable(S),
      subsume(V1,V2).
```

Executing the program on the example input gives us the following output.

```
microclones((b1,b1),(a1,a1))
microclones((body1,e1),(body2,b1))
microclones((body2,b1),(body1,a1))
microclones((body2,body2),(body0,body0))
microclones((body2,e1),(body1,a1))
microclones((c1,c1),(b1,b1))
microclones((e1,e1),(a1,a1))
microclones((e1,e1),(b1,b1))
```

Again, this output matches the microclone relations shown in Figure 3.4.

3.5.5 Encoding Cloning Process

Encoding the Extended Set of Edges

Predicate may_have

We used predicate $may\_have((X,S,V),C,(Y,F))$ to indicate that we must consider edge (Y,S,V) in the extended set of edges (from X in the context C), since X in C is microcloned from Y in F. This is equivalent to $N_P(A,S)$ in Definition 27, where A is the set of instances that X is microcloned from, and P is the set of graphs $G^*(F)$ (C is microcloned from F).

a16:

```
% X may have value V from Y if: X microclones value V from
   some instance Y
% may_have((X,S,V),C,(Y,F)): we may have (X,S,V) in C from Y
   in F
may_have((X,S,V),C,(Y,F)) :-
      hasc((Y,S,V),F),
      microclones((X,C), (Y,F)).
```

Executing the program on the example input gives us the following output.

```
may_have((b1,has_part,body1),b1,(a1,a1))
may_have((b1,has_part,head1),b1,(a1,a1))
may_have((body1,has_part,four_legs),e1,(body2,b1))
may_have((body1,has_part,fur1),e1,(body2,b1))
may_have((body1,has_size,big1),e1,(body2,b1))
may_have((body2,has_part,four_legs),b1,(body1,a1))
may_have((body2,has_part,four_legs),e1,(body1,a1))
may_have((body2,has_size,big1),b1,(body1,a1))
may_have((body2,has_size,big1),e1,(body1,a1))
may_have((c1,has_part,body2),c1,(b1,b1))
may_have((c1,has_part,head1),c1,(b1,b1))
may_have((e1,has_part,body1),e1,(a1,a1))
may_have((e1,has_part,body2),e1,(b1,b1))
may_have((e1,has_part,head1),e1,(a1,a1))
may_have((e1,has_part,head1),e1,(b1,b1))
```

Predicate may_not_have

Rule $a17$ encodes the instances that are subsumed by a node in $G(C)$.

a17:

```
% Result of set difference with respect to subsume: KB has the edge
   (X, S, V1), may have (X, S, V2) from Xj but V1 subsume V2.
```

```
% V is in N \_{subsume} M if may_have ((X,S,V), C,(Y,F)) and not
   may_not_have((X, S, V), C)
may_not_have((X, S, V2), C) :-
      has(X, S, V1), may_have((X, S, V2), C, (Xj, F)),
      subsume(V1, V2),
      X != Xj,
      V1 != V2.
```

Executing the program on the example input gives us the following output.

```
may_not_have((b1,has_part,body1),b1)
```

This result is expected because our first principle of unification that makes us select *body*2.

Predicate equ

Rule $a18$ encodes that $V1$ and $V2$ are equally specific nodes when they subsume each other; $(X,S,V1)$ and $(X,S,V2)$ are two possible edges from X in context C. The condition *not may_not_have*$((X,S,V1),C)$ is to remove the nodes $V1$ that are subsumed by a node in $G(C)$. This is equivalent to obtaining the set $N \setminus_{subsume} M$ in Definition 27. Note that the equivalent relation between nodes are transitive and symmetric. In a set of equally specific nodes, every node has $equ()$ relation to every other node.

a18:

```
% Equally specific nodes
equ((X, S, V1), (Xi, Fi), (X, S, V2), (Xj, Fj), C) :-
      may_have((X, S, V1), C, (Xi, Fi)), may_have((X, S, V2), C, (Xj
         , Fj)),
      subsume(V2, V1), subsume(V1, V2),
      not may_not_have((X, S, V1), C), not may_not_have((X, S, V2),
         C),
      Xi != X, Xj != X, Xi != Xj.
```

Executing the program on the example input gives us the following output.

```
equ((e1,has_part,body1),(a1,a1),(e1,has_part,body2),(b1,b1),e1)
equ((e1,has_part,body2),(b1,b1),(e1,has_part,body1),(a1,a1),e1)
equ((e1,has_part,head1),(a1,a1),(e1,has_part,head1),(b1,b1),e1)
equ((e1,has_part,head1),(b1,b1),(e1,has_part,head1),(a1,a1),e1)
```

Predicate may_not_have2

Rule $a19$ encodes instances that are less specific than another instance. If two instances $x4$ and $x5$ subsume each other like in Figure 3.3, and they are not subsumed by any other node, they would not be included in $not\_may\_have2()$.

a19:

```
% Most specific nodes: nodes that are not subsumed by any node,
   except its equally specific ones.
may_not_have2((X, S, V1), C) :-
      may_have((X, S, V1), C, (Xi,_)), may_have((X, S, V2), C, (Xj,
         _)),
      subsume(V2, V1), not_subsume(V1, V2),
      V1 != V2, Xi != X, Xj != X.
```

Executing the program on the example input gives us nothing on may_not_have2 since our example $a1$-$e1$ does not have this case.

Predicate edges_s

Rule $a20$ encodes the most specific nodes (nodes in $MIN_{subsume}(N_P(A,S) \setminus_{subsume} N_{G(C)}(x,S))$) of Definition 27. When a node V is not less specific than any node, either V or one of its equally specific nodes will be randomly chosen and put in the $edges\_s()$.

a20:

```
      1 { edges_s((X, S, V1), C, (Xj, Fj)): equ((X, S, V), (Xi, Fi),
          (X, S, V1), (Xj, Fj), C);
  edges_s((X, S, V), C, (Xi, Fi))} 1 :-
```

```
may_have((X, S, V), C, (Xi, Fi)),
not may_not_have((X, S, V), C),
not may_not_have2((X, S, V), C).
```

Following is one output of the program with respect to the example input.

```
edges_s((b1,has_part,head1),b1,(a1,a1))
edges_s((body1,has_part,four_legs),e1,(body2,b1))
edges_s((body1,has_part,fur1),e1,(body2,b1))
edges_s((body1,has_size,big1),e1,(body2,b1))
edges_s((body2,has_part,four_legs),b1,(body1,a1))
edges_s((body2,has_part,four_legs),e1,(body1,a1))
edges_s((body2,has_size,big1),b1,(body1,a1))
edges_s((body2,has_size,big1),e1,(body1,a1))
edges_s((c1,has_part,body2),c1,(b1,b1))
edges_s((c1,has_part,head1),c1,(b1,b1))
edges_s((e1,has_part,body1),e1,(a1,a1))
edges_s((e1,has_part,head1),e1,(b1,b1))
```

The output tells that we need to add to $G^*(b1)$:

1. The edge $(b1, has\_part, head1)$, which is obtained from $G(a1)$ in $G^*(a1)$.

2. The edges $(body2, has\_part, four\_legs)$ and $(body2, has\_size, big1)$, which are obtained from $G(body1)$ in $G^*(a1)$.

This result exactly matches the one in Figure 3.8, which we analyzed previously.

Similarly, we need to add two edges to $G^*(c1)$ and seven edges to $G^*(e1)$, which also match previous results. Note that the rest of the graph $G(body2)$ in $G^*(b1)$ is also added to $G^*(c1)$ but through rule $a27$. It also indicates that no new edge is added to $G^*(d1)$, keeping $G^*(d1)$ indifferent from $G(d1)$

Encoding the Connectivity in Object Graph

We encode an edge $((X,S,Y))$ in the context of C by $hasc((X,S,Y),C)$. Using predicate $connectI(X,Y,C)$, we encode the connectivity through inheritable edges from X to Y in the context C. $connect(X,Y,C)$ is used to encode the connectivity through $connectI$ and then through an optional non-inheritable edge.

a21, a22:

```
% ConnectI from X to Y if there is a chain of inheritable
    edges from X to Y
connectI(X, X, C) :- ins(X), ins(C).
connectI(X, Z, C) :- connectI(Y,Z,C), hasc((X, S, Y), C), not
    noninheritable(S).
```

a23, a24:

```
% Connect from X to Y if there connectI there is a chain of
    edges from X to Y; the optional last edge is noninheritable
    .
connect(X, Y, C) :- connectI(X, Y, C).
connect(X, Z, C) :- connectI(X,Y,C), hasc((Y, S, Z), C),
    noninheritable(S).
```

Executing the program on the example input gives us the following output. Only the first 10 predicates of *connectI* and *connect* from $a1$-$e1$ are shown. As we can see, $a1$ *connectI*s to $a1$ in all the of the contexts $a1$, $b1$, $big1$, etc. *connect* also links instance (i.e. $d1$) to the classes (i.e. *body*).

```
connectI(a1,a1,a1)
connectI(a1,a1,b1)
connectI(a1,a1,big1)
connectI(a1,a1,body0)
connectI(a1,a1,body1)
```

```
connectI(a1,a1,body2)
connectI(a1,a1,c1)
connectI(a1,a1,d1)
connectI(a1,a1,e1)
connectI(a1,a1,four_legs)
connectI(a1,a1,fur1)
connectI(a1,a1,head1)

connectI(b1,b1,a1)
connectI(b1,b1,b1)
connectI(b1,b1,big1)
connectI(b1,b1,body0)
connectI(b1,b1,body1)
connectI(b1,b1,body2)
connectI(b1,b1,c1)
connectI(b1,b1,d1)
connectI(b1,b1,e1)
connectI(b1,b1,four_legs)

connectI(c1,big1,c1)
connectI(c1,body2,c1)
connectI(c1,c1,a1)
connectI(c1,c1,b1)
connectI(c1,c1,big1)
connectI(c1,c1,body0)
connectI(c1,c1,body1)
connectI(c1,c1,body2)
connectI(c1,c1,c1)
connectI(c1,c1,d1)

connectI(d1,big1,b1)
```

```
connectI(d1,big1,c1)
connectI(d1,big1,e1)
connectI(d1,body2,a1)
connectI(d1,body2,b1)
connectI(d1,body2,big1)
connectI(d1,body2,body0)
connectI(d1,body2,body1)
connectI(d1,body2,body2)
connectI(d1,body2,c1)

connectI(e1,big1,e1)
connectI(e1,body1,e1)
connectI(e1,e1,a1)
connectI(e1,e1,b1)
connectI(e1,e1,big1)
connectI(e1,e1,body0)
connectI(e1,e1,body1)
connectI(e1,e1,body2)
connectI(e1,e1,c1)
connectI(e1,e1,d1)

connect(a1,a1,a1)
connect(a1,animal,a1)
connect(a1,big1,a1)
connect(a1,body,a1)
connect(a1,body1,a1)
connect(a1,four_legs,a1)
connect(a1,head,a1)
connect(a1,head1,a1)
connect(a1,legs,a1)
connect(a1,size,a1)
```

```
connect(b1,b1,a1)
connect(b1,b1,b1)
connect(b1,b1,big1)
connect(b1,b1,body0)
connect(b1,b1,body1)
connect(b1,b1,body2)
connect(b1,b1,c1)
connect(b1,b1,d1)
connect(b1,b1,e1)
connect(b1,b1,four_legs)

connect(c1,c1,a1)
connect(c1,c1,b1)
connect(c1,c1,big1)
connect(c1,c1,body0)
connect(c1,c1,body1)
connect(c1,c1,body2)
connect(c1,c1,c1)
connect(c1,c1,d1)
connect(c1,c1,e1)
connect(c1,c1,four_legs)

connect(d1,big1,b1)
connect(d1,big1,c1)
connect(d1,big1,e1)
connect(d1,body,a1)
connect(d1,body,b1)
connect(d1,body,big1)
connect(d1,body,body0)
connect(d1,body,body1)
```

```
connect(d1,body,body2)
connect(d1,body,c1)


connect(e1,a1,a1)
connect(e1,a1,b1)
connect(e1,a1,big1)
connect(e1,a1,body0)
connect(e1,a1,body1)
connect(e1,a1,body2)
connect(e1,a1,c1)
connect(e1,a1,d1)
connect(e1,a1,e1)
connect(e1,a1,four_legs)
```

Encoding the Construction of Unified Object Graph

$G^*(C)$ is encoded by all the predicates $hasc((X,S,V),C)$, where C connects to X in the context of C, represented by $connect(C,X,C)$.

We add new edges to the $G*(C)$ by adding new $hasc((X,S,V),C)$. Using rules $a25$, $a26$ and $a27$. we encode $G^*(C)$ according to Definition 28.

Rule $a25$ adds all of the existing edges of $G(C)$ to $G^*(C)$.

Rule $a26$ puts all edges in the extended set of edges into $G^*(C)$. The extended set of edges (Definition 27) is encoded by predicate $edges_s((X,S,V),C,(Xi,F))$.
$edges_s((X,S,V),C,(Xi,F))$ means that the edges (Xi,S,V) from Xi in F is in the extended set of edges from X in C.

Rule $a27$ puts every edge in $G(V)$ in $G^*(F)$ into $G^*(C)$, where C is g-cloned-from F, and V was added into $G^*(C)$ by rule $a26$.

a25, a26, a27:

```
% Add all the existing nodes from x
```

```
hasc((X, S, V), C) :-
      ins(C),
      has(X, S, V).

% Add the edge from x to the most specific nodes
hasc((X, S, V), C) :-
      edges_s((X, S, V), C, (Xi, F)).

% Add all other nodes in G(V) in G*(F) if (X,S,V) is added
   through edges_s( )
hasc((Z1, SZ, Z2), C) :-
      edges_s((X, S, V), C, (Xi, F)),
      hasc((Z1, SZ, Z2), F),
      connect(V, Z2, F),
      connect(V, Z1, F).
```

Executing the program on the example input gives us the following output. Here, we show only the facts with respect to the context of $a1$ and $b1$.

```
hasc((a1,has_part,body1),a1)
hasc((a1,has_part,head1),a1)
hasc((a1,instance_of,animal),a1)
hasc((a1,prototype_of,animal),a1)
hasc((b1,has_part,body2),a1)
hasc((b1,instance_of,mammal),a1)
hasc((b1,prototype_of,mammal),a1)
hasc((big1,instance_of,size),a1)
hasc((body0,instance_of,body),a1)
hasc((body1,has_part,four_legs),a1)
hasc((body1,has_size,big1),a1)
hasc((body1,instance_of,body),a1)
hasc((body2,cloned_from,body0),a1)
```

```
hasc((body2,has_part,fur1),a1)
hasc((body2,instance_of,body),a1)
hasc((c1,cloned_from,b1),a1)
hasc((c1,has_part,tail1),a1)
hasc((c1,instance_of,mutated_dog),a1)
hasc((d1,has_part,body2),a1)
hasc((d1,instance_of,mutated_animal),a1)
hasc((d1,prototype_of,mutated_animal),a1)
hasc((e1,cloned_from,a1),a1)
hasc((e1,cloned_from,b1),a1)
hasc((e1,instance_of,animal),a1)
hasc((four_legs,instance_of,legs),a1)
hasc((fur1,instance_of,fur),a1)
hasc((head1,instance_of,head),a1)
hasc((mammal,superclass,animal),a1)
hasc((tail1,instance_of,tail),a1)

hasc((a1,has_part,body1),b1)
hasc((a1,has_part,head1),b1)
hasc((a1,instance_of,animal),b1)
hasc((a1,prototype_of,animal),b1)
hasc((b1,has_part,body2),b1)
hasc((b1,has_part,head1),b1)
hasc((b1,instance_of,mammal),b1)
hasc((b1,prototype_of,mammal),b1)
hasc((big1,instance_of,size),b1)
hasc((body0,instance_of,body),b1)
hasc((body1,has_part,four_legs),b1)
hasc((body1,has_size,big1),b1)
hasc((body1,instance_of,body),b1)
hasc((body2,cloned_from,body0),b1)
```

```
hasc((body2,has_part,four_legs),b1)
hasc((body2,has_part,fur1),b1)
hasc((body2,has_size,big1),b1)
hasc((body2,instance_of,body),b1)
hasc((c1,cloned_from,b1),b1)
hasc((c1,has_part,tail1),b1)
hasc((c1,instance_of,mutated_dog),b1)
hasc((d1,has_part,body2),b1)
hasc((d1,instance_of,mutated_animal),b1)
hasc((d1,prototype_of,mutated_animal),b1)
hasc((e1,cloned_from,a1),b1)
hasc((e1,cloned_from,b1),b1)
hasc((e1,instance_of,animal),b1)
hasc((four_legs,instance_of,legs),b1)
hasc((fur1,instance_of,fur),b1)
hasc((head1,instance_of,head),b1)
hasc((mammal,superclass,animal),b1)
hasc((tail1,instance_of,tail),b1)
```

3.5.6 Correctness of the ASP Encoding

Definition 29 (ASP program) *Given a knowledge base KB, the ASP program Π_{KB} is the answer set program consisting of the following:*

1. *The rules from a*1 *to a*27.

2. *All facts of the form has(X,S,V) and class(C) that are either in the class hierarchy graph or in the object graphs.*

Lemma 11 *X is an in a knowledge base KB iff:*

$$\Pi_{KB} \models ins(X)$$

The proof for lemma 11 will be given in the Appendix.

Lemma 12 *X is a tc-instance-of Y with respect to a knowledge base KB iff:*

$$\Pi_{KB} \models tc\_instance\_of(X,Y)$$

The proof of lemma 12 will be given in the Appendix.

Lemma 13 *X subsumes Y with respect to a knowledge base KB iff:*

$$\Pi_{KB} \models subsume(X,Y)$$

Lemma 14 *X is g-cloned-from Y with respect to a knowledge base KB iff:*

$$\Pi_{KB} \models g\_cloned\_from(X,Y)$$

The proof of lemma 14 will be given in the Appendix.

Lemma 15 *X in C is microcloned from Y in F with respect to a knowledge base KB iff:*

$$\Pi_{KB} \models microclones((V1,C),(V2,F))$$

Proposition 3 *Let X and Y be two instances in a knowledge base KB. Y is in $G^*(X)$ iff:*

$$\Pi_{KB} \models connect(X,Y,X)$$

The proof of Proposition 3 will be given in the Appendix.

3.6 Efficient Unification

3.6.1 Efficient ASP Implementation

In the ASP implementation above, the program will do unification for all instances with respect to the context of all the instances in the KB. This process could take forever on a large KB such as AURA. However, among various kinds of questions possibly posed to a

knowledge base, many can be more efficiently answered by modifying some of the rules given in the earlier section. For example, we only want to accomplish unification with respect to the context of several instances and for some instances related to the context instances; most of instances in the KB are irrelevant to the questions and will be ruled out. This is similar to the use of magic sets in efficiently answering Datalog queries, whereas Datalog rules are transformed based on query patterns. In the following, we show how some rules can be modified for efficiency purposes without sacrificing correctness for the specific kind of queries.

We choose two types of queries for illustration. For questions like **"What is X?"**, **"What is the difference between X and Y?"**, we only need to have complete information about X and Y(i.e. $G^*(X)$ and $G^*(Y)$). That means we have to unify $G(X)$ in context of X and $G(Y)$ in the context of Y. This would need $G^*(X_i)$s and $G^*(Y_j)$s; X_is and Y_js are nodes of which X and Y are g-cloned from (Definition 28), meaning that we have to unify $G(X_i)$s and $G(Y_j)$s in the context of X_is and Y_js.

So if we want to compute the set S of context instances, initially, we add X and Y to S. In the second step, we add X_is and Y_js, which X and Y are g-cloned from. We then add instances of which X_is and Y_js themselves are g-cloned from, and so on.

We use predicate $context(X)$ to encode that we need to use X as a context instance. Rule $a28$ finds all of the needed context instances.

a28:

```
context(Y) :-     context(X), g_cloned_from(X,Y).
```

From the unification definitions above, it is also clear that for computing the unified $G^*(X)$, we only need the instances in $G(X)$, $G(X_i)$s, and $G(M_k)$s (X_is are g-cloned from M_k),... In other words, we use only the instances in $G(T)$ where T is one of the context that we need.

Similarly to $connectI(X,Y,C)$ and $connect(X,Y,C)$ in $a21$, $a22$, $a23$ and $a24$, we use predicate $relateI(C,X)$ to encode connectivity through inheritable edges from C to X in $G(C)$; $relate(C,X)$ for connectivity through $relateI$ and then through an optional non-inheritable edge.

Compared to rules $a21$, $a22$, $a23$ and $a24$, the rules $a29$, $a30$, $a31$, and $a32$ have some differences:

1. They use the edges in the original object graphs (encoded by $has(X,S,Y)$) instead of the unified object graphs (encoded by $hasc((X,S,Y),C)$).

2. $relateI(C,X)$s and $relate(C,X)$s are defined only when $context(C)$ is true because we only care about instances in $G(T)$ where T is one of the context that we need.

a29, a30, a31, a32:

```
relateI(X, X) :- context(X).
relateI(X, Z) :-
      context(X),
      relateI(X,Y),
      has(Y, S, Z),
      not noninheritable(S).

relate(X,Y) :-
      context(X),
      relateI(X,Y).
relate(X, Z) :-
      context(X),
      relateI(X,Y),
      has(Y, S, Z),
      noninheritable(S).
```

We then limit ourselves to the instances of interest by modifying rules $a2$ and $a3$.

We change

$a2$, $a3$:

```
% grab all instances
ins(X) :- has(X,S,V), not class(X).
ins(V) :- has(X,S,V), not class(V).
```

to

a33, a34:

```
ins(X) :- has(X,S,V), not class(X),
      context(C), relate(C, X).
ins(V) :- has(X,S,V), not class(V),
      context(C), relate(C, X).
```

For other rules: $a13$-$a15$, $a16$-$a20$, $a21$-$a22$, $a23$-$a24$, and $a25$-$a27$, we add rule $context(C)$ (or $context(F)$, $context(X1)$, $context(X2)$) where C is used as the context of unification in that rule similar to the way rule $a2$ was changed to $a33$.

Similarly to how $a2$ was changed to $a33$, in rule $a25$, we also add $relate(C,X)$ because we then only consider existing edges in $G(C)$ to $G^*(C)$. We can see all those rules ($a13$-$a15$, $a16$-$a20$, $a21$-$a22$, $a23$-$a24$ and $a25$-$a27$) after the modification by uncommenting the commented atoms in each rule shown before. For example, $a13$ was previously like this:

$a13$:

```
microclones((X2,X2), (X1,X1)) :-
      % context(X1), context(X2),
      ins(X1), ins(X2),
      g_cloned_from(X2,X1).
```

After uncommenting "% context(X1), context(X2)", we get the modified rule:

a35:

```
microclones((X2,X2), (X1,X1)) :-
      context(X1), context(X2),
      ins(X1), ins(X2),
      g_cloned_from(X2,X1).
```

The complete program with all the modified rules is listed in the Appendix.

3.6.2 Query Execution Timings

We tested the rules on a corpus that contains 5,180 classes, 32,339 instances, and 5,673 entries in the class hierarchy graph, and 198,301 entries in the various object graphs. The running time with the efficient encoding is around 3 seconds per question, while the earlier encoding ran for more than two hours without giving an answer.

3.6.3 Correctness of the ASP Encoding

Definition 30 (ASP program) *Given a knowledge base KB, the ASP program Π'_{KB} is the answer set program consisting of all the following:*

1. *The modified rules of $a1$ - $a27$.*
2. *All facts of the form $has(X,S,V)$ and $class(C)$ that are either in the class hierarchy graph, or in the object graphs.*
3. *The rules $a28$ and $a29$-$a32$.*
4. *The facts of the form $context(C)$ of all the object graphs $G(C)$s of interest.*

Lemma 16 *X is in the object graph $G(C)$ in a knowledge base KB iff:*

$$\Pi'_{KB} \models relate(C,X)$$

The proof of Lemma 16 will be given in the Appendix.

Proposition 4 *Let X and Y be two instances in a knowledge base KB. Y is in $G^*(X)$ iff:*

$$\Pi'_{KB} \models connect(X,Y,X)$$

The proof sketch of Proposition 4 will be given in the Appendix.

3.7 Encoding of Question Answering

The AURA system supports seven types of questions, namely:

1. Identifying superclasses.
2. Giving an example of a class.
3. Computing a slot value.
4. Describing a class.
5. Checking if an assertion is true or false.
6. Comparing individuals.
7. Computing the relationship between two individuals.

Our current ASP based system can answer all types of questions supported by AURA. In this section, we show some of the comparatively difficult ones (We do not show the ASP rules for "Identifying superclasses" and "Giving an example of a class," as they are very straight forward in our ASP implementation).

3.7.1 Describe a Class

The most simple yet fundamental question is: "What is (a/an) X?". There are two ways to answer this question:

1. Return the full list of properties for X.

2. Return the most distinguishable properties of X, in comparison to its immediate superclass.

The first way provides a direct answer, which is an object graph $G^*(X)$. The output of the unification process of $G(X)$ in the context of X contains the facts of $G^*(X)$ and many other facts. However, we can get all the nodes of $G^*(X)$ from the facts of the form $connect(X,Y,X)$. $connect(X,Y,X)$ must be true for each node Y in $G^*(X)$.

Return the Full List of Properties for X

The question "What is (a/an) X?" is encoded like the following. Here, X is the **animal** class.

```
% Q: What is an animal?
question(q1).
qhas(q1, type, what).
qhas(q1, cat, is).
qhas(q1, param1, animal).
```

Following are the two rules utilized to answer the question. In both rules, we first find the instance C of the class X in the question. Rule $a36$ puts the fact $context(C)$ so that the unification process can start. Rule $a37$ returns all the edges from M to N in $G^*(C)$. M must be connected from C through inheritable edges because if it is not, it must be reached through a non-inheritable edge; it is a leaf node of the object graph by definition. $G^*(C)$ should not contain any edge from such leaf nodes.

a36, a37:

```
%     A: [answer by return the full list of properties for
   cell.]
context(C) :- question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, is),
```

```
        qhas(Q, param1, Class),
        has(C, prototype_of, Class).

answer(Q, hasc((M, S, N), C)) :- hasc((M, S, N), C),
        context(C),
        connectI(C, M, C),
        connect(C, N, C),
        qhas(Q, type, what),
        qhas(Q, cat, is),
        qhas(Q, param1, Class),
        has(C, prototype_of, Class).
```

Executing the answering rules on our example KB with the encoding "What is an animal?" give us the $G^*(a1)$ in the following:

```
answer(q1,hasc((a1,has_part,body1),a1))
answer(q1,hasc((a1,has_part,head1),a1))
answer(q1,hasc((a1,instance_of,animal),a1))
answer(q1,hasc((a1,prototype_of,animal),a1))
answer(q1,hasc((big1,instance_of,size),a1))
answer(q1,hasc((body1,has_part,four_legs),a1))
answer(q1,hasc((body1,has_size,big1),a1))
answer(q1,hasc((body1,instance_of,body),a1))
answer(q1,hasc((four_legs,instance_of,legs),a1))
answer(q1,hasc((head1,instance_of,head),a1))
context(a1)
```

It only needs the context of $a1$ in order to answer, and the answer has the correct edges, matching with the examples previously shown. Similarly, we can verify the answer of "What is a mammal?," which is $G^*(b1)$ in the following output. Both $a1$ and $b1$ were used as the context for unification.

```
answer(q1,hasc((b1,has_part,body2),b1))
answer(q1,hasc((b1,has_part,head1),b1))
answer(q1,hasc((b1,instance_of,mammal),b1))
answer(q1,hasc((b1,prototype_of,mammal),b1))
answer(q1,hasc((big1,instance_of,size),b1))
answer(q1,hasc((body2,cloned_from,body0),b1))
answer(q1,hasc((body2,has_part,four_legs),b1))
answer(q1,hasc((body2,has_part,fur1),b1))
answer(q1,hasc((body2,has_size,big1),b1))
answer(q1,hasc((body2,instance_of,body),b1))
answer(q1,hasc((four_legs,instance_of,legs),b1))
answer(q1,hasc((fur1,instance_of,fur),b1))
answer(q1,hasc((head1,instance_of,head),b1))
context(a1)
context(b1)
```

Next, we run another example on the real AURA KB. The question is "What is a cell?". The answer is the $G^*(cell325)$, where $cell325$ is an prototype of *cell*. Following are 10-lines for 10 edges from a total of the 648 edges of $G^*(cell325)$. Compared to the original $G(cell325)$, $G^*(cell325)$ was obtained from two more classes ($cell18399$ and $tangible\_entity22850$) and was doubled in size (to 648 from 229).

```
answer(q1,hasc((cell325,clean_instance_of,cell),cell325))
answer(q1,hasc((cell325,cloned_from,cell18399),cell325))
answer(q1,hasc((cell325,cloned_from,tangible_entity22850),
   cell325))
answer(q1,hasc((cell325,diameter,length_value15324),cell325))
answer(q1,hasc((cell325,does_not_enclose,
   extra_cellular_matrix11481),cell325))
answer(q1,hasc((cell325,has_part,chromosome10040),cell325))
answer(q1,hasc((cell325,has_part,cytoplasm689),cell325))
answer(q1,hasc((cell325,has_part,enzyme6401),cell325))
```

```
answer(q1,hasc((cell325,has_part,plasma_membrane14508),cell325
    ))
answer(q1,hasc((cell325,has_part,ribosome23653),cell325))
```

Return the Most Distinguishable Properties of X

In this section, we present another way to answer the question. Similar to previous method, in order to answer the question "What is a/an X?", we first find the prototype P of class X and return the properties of P. Note that $G^*(P)$ was created from $G(P)$ by obtaining additional information from three sources:

1. Prototypes of X's ancestor classes
2. Instances that P is cloned from (encoded by cloned-from edge)
3. Other prototypes of P's classes (including X)

$G(P)$ contains the most distinguishable properties of P compared to those sources, so $G(P)$ itself is already a reasonable answer to the question. But if we want to limit the comparisons to ancestor classes, we can modify the g-cloned-from rules ($a10$-$a12$) so that $G(P)$ does not use additional information from the prototypes of X's ancestor classes. The $G^*(P)$ returned from the answering rules $a36$, $a37$ would return simpler answers than those of the previous section.

Another way to answer this question is by showing the differences between P and a prototype of X's superclass(es). Details of how to show the differences will be discussed in next section.

To summarize, we can consider three ways of returning the most distinguishable properties of X:

1. Return $G(P)$.

2. Return $G^*(P)$ after modifying rules $a10$-$a12$.

3. Return the differences between P and a prototype of X's superclass(es).

Here, we show the ASP rules for the first and second ways. The last one will be discussed in next section.

Return $G(P)$

Following is the modified version of rule $a37$ to return $G(P)$ instead $G^*(P)$. , we still need to use the rule $a36$.

Existing rule $a36$**:**

```
%     A: [answer by return the full list of properties for
   cell.]
context(C) :- question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, is),
      qhas(Q, param1, Class),
      has(C, prototype_of, Class).
```

a38:

```
answer(Q, has(X, S, Y)) :- has(X, S, Y),
      relateI(P, X),
      relate(P, Y),
      qhas(Q, type, what),
      qhas(Q, cat, is),
      qhas(Q, param1, Class),
      has(P, prototype_of, Class).
```

Rule $a38$ returns all the edges in $G(P)$, compared to $a37$, which uses *relate* and *relateI* instead of *connectI* and *connect*.

Modifying rules $a10$**-**$a12$

In the following section, we show the change in rules $a10$-$a12$ so that $G^*(P)$ does not use

additional information from the prototypes of X's ancestor classes. In fact, we only need to change $a12$; instead of $tc\_instance\_of$, we need to use $instance\_of$.

Existing rules $a10$ and $a11$:

```
g_cloned_from(X,Y) :- X!=Y, has(X,cloned_from,Y).
g_cloned_from(X,Y) :- X!=Y, has(X,clone_built_from,Y).


g_cloned_from(X,Y) :- X!=Y, instance_of(X,M), has(Y,
    prototype_of,M).
```

Executing the answering rules ($a36$ and $a37$) with modified rule $a39$ on our example KB with the encoding "What is a mammal?" give us the $G^*(b1)$ in the following:

```
answer(q1,hasc((b1,has_part,body2),b1))
answer(q1,hasc((b1,instance_of,mammal),b1))
answer(q1,hasc((b1,prototype_of,mammal),b1))
answer(q1,hasc((body2,cloned_from,body0),b1))
answer(q1,hasc((body2,has_part,fur1),b1))
answer(q1,hasc((body2,instance_of,body),b1))
answer(q1,hasc((fur1,instance_of,fur),b1))
context(b1)
```

Since $a1$ is the prototype of *animal*, it was not used to unify. $G^*(b1)$ now has 7 edges (compared to 13 of the previous).

3.7.2 Comparing Individuals

This is another fundamental question, which asks for the similarities and differences between two individuals. The question could be about the specific instances such as "What are the similarities between $b1$ and $c1$?", but this is uncommon. Most of the time, the questions are about classes such as "What are the differences between prokaryotic and eukaryotic cells?"

We first present a method in answering questions about similarities and later extend it to the "differences" case.

Class Similarities

As usual, we begin by encoding the question. The question "What are the similarities between *Class*1 and *Class*2?" is encoded as the following:

aa390:

```
% Q: What are the similarities between Class1 and Class2?''
question(q2).
qhas(q2, type, what).
qhas(q2, cat, similarity).
qhas(q2, param1, class1).
qhas(q2, param2, class2).
```

For example, the question "What are the similarities between mutated animal and mammal?" is encoded as follows. *Class*1 and *Class*2 here are **mutated animal** and **mammal** respectively.

Example 22

```
% Q: What are the similarities between mutated_animal
    and mammal?''
question(q2).
qhas(q2, type, what).
qhas(q2, cat, similarity).
qhas(q2, param1, mammal).
qhas(q2, param2, mutated_animal).
```

Similarly to answering the previous question, we must find the prototypes $C1$ and $C2$ of *class*1 and *class*2 respectively; then, we add the facts $context(C1)$ and $context(C2)$ so the unification may begin. Note that $context(C1;C2)$ is the shortcut for $context(C1)$ and $context(C2)$.

a41:

```
context(C1;C2) :- question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, similarity),
      qhas(Q, param1, Class1),
      qhas(Q, param2, Class2),
      has(C1, prototype_of, Class1),
      has(C2, prototype_of, Class2).
```

If the question is about *mammal* and *mutated_animal* in our example KB, $b1$ and $d1$ respectively are the prototypes of *mammal* and *mutated_animal*. So, the answer set would contain $context(d1)$ and $context(b1)$.

In rule $a42$, we use predicate $compare(Q,(C1,C1),(C2,C2))$ to see that to answer question Q, we need to compare $C1$ with $C2$ (more precisely, $C1$ in the context of $C1$ with $C2$ in $C2$).

a42:

```
% We need to compare C1 in the context of C1 with C2 in C2
compare(Q,(C1, C1), (C2, C2)) :- question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, similarity),
      qhas(Q, param1, Class1),
      qhas(Q, param2, Class2),
      has(C1, prototype_of, Class1),
      has(C2, prototype_of, Class2).
```

To answer this, we introduce the predicate $sim((M,X),(N,Y),S,V)$ to stand for the same value V of the same slot S shared by two instances M in X and N in Y. The rules for this predicate are as follows:

a43:

```
% (M in X) and (N in Y) both contain V for slot S
sim((M,X),(N,Y),S,V) :- compare(Q,(M,X), (N,Y)),
      hasc((M,S,V), X),
      hasc((N,S,V1),Y),
      V==V1.
```

Let us return to our *mammal* vs. *mutated_animal*; in the answer set, we would have $sim((d1,d1),(b1,b1),has\_part,body2)$ since we need to compare $d1$ with $b1$, and each of them has-part $body2$. However, as mentioned earlier, the contents of $body2$ in $b1$ and in $d1$ are not the same. We would need further comparison in $G(body2)$. Rule $a44$ encodes this logic. We will continue to compare V in X and V in Y if both M in X and N in Y contain the edge of type S to V.

a44:

```
compare(Q, (V, X), (V, Y)) :- compare(Q,(M,X), (N,Y)),
      sim((M,X),(N,Y),S,V),
      connectI(X, V, X),
      connectI(Y, V, Y).
```

Using this rule in our example, we have the following output. It continued to compare $body2$ in $d1$ and $body2$ in $b1$ and found that both of them have $fur1$; it then continued comparing $fur1$ in $d1$ and $fur1$ in $d1$.

```
compare(q2,(body2,d1),(body2,b1))
compare(q2,(d1,d1),(b1,b1))
compare(q2,(fur1,d1),(fur1,b1))
sim((body2,d1),(body2,b1),cloned_from,body0)
sim((body2,d1),(body2,b1),has_part,fur1)
sim((body2,d1),(body2,b1),instance_of,body)
sim((d1,d1),(b1,b1),has_part,body2)
sim((fur1,d1),(fur1,b1),instance_of,fur)
```

Finally, the answer of the question is obtained simply by combing all *sim*() predicates.

a45:

```
answer(Q, sim((M,X),(N,Y),S,V)) :-  compare(Q,(M,X), (N,Y)),
      question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, similarity),
      sim((M,X),(N,Y),S,V).
```

Following is the output of answering question, "What are the similarities between mutated animal and mammal?". The program we used is the combination of : Π_{KB}, question encoding rules $a40$, rules $a41$, $a42$, $a43$, $a44$ and $a45$.

```
answer(q2,sim((body2,d1),(body2,b1),cloned_from,body0))
answer(q2,sim((body2,d1),(body2,b1),has_part,fur1))
answer(q2,sim((body2,d1),(body2,b1),instance_of,body))
answer(q2,sim((d1,d1),(b1,b1),has_part,body2))
answer(q2,sim((fur1,d1),(fur1,b1),instance_of,fur))
compare(q2,(body2,d1),(body2,b1))
compare(q2,(d1,d1),(b1,b1))
compare(q2,(fur1,d1),(fur1,b1))
context(a1)
context(b1)
context(d1)
sim((body2,d1),(body2,b1),cloned_from,body0)
sim((body2,d1),(body2,b1),has_part,fur1)
sim((body2,d1),(body2,b1),instance_of,body)
sim((d1,d1),(b1,b1),has_part,body2)
sim((fur1,d1),(fur1,b1),instance_of,fur)
```

Class Differences

Here we discuss how to answer questions like: "What are the differences between prokaryotic and eukaryotic cells?". This question is encoded exactly like the one about similarity, except the question category (encoded by $qhas(Q, cat, CATEGORY)$ is changed to $qhas(Q, cat, difference)$:

```
% Q: What are the differences between mutated_animal and
   mammal?''
question(q2).
qhas(q2, type, what).
qhas(q2, cat, difference).
qhas(q2, param1, mammal).
qhas(q2, param2, mutated_animal).
```

Similarly, we need to change "similarities" to "differences" in rules $a41$ and $a42$ to make rules $a46$ and $a47$.

a46, a47:

```
% Question about differences
% ------------------------------
context(C1;C2) :- question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, difference),
      qhas(Q, param1, Class1),
      qhas(Q, param2, Class2),
      has(C1, prototype_of, Class1),
      has(C2, prototype_of, Class2).

% We need to compare C1 in the context of C1 with C2 in C2
compare(Q,(C1, C1), (C2, C2)) :- question(Q),
      qhas(Q, type, what),
```

```
        qhas(Q, cat, difference),
        qhas(Q, param1, Class1),
        qhas(Q, param2, Class2),
        has(C1, prototype_of, Class1),
        has(C2, prototype_of, Class2).
```

For differences, we define "difference" as: **What instance X has but Y doesn't, and vice versa.** We use the predicate $diff((M,X),(N,Y),S,V,h1)$ which stands for: M in X has V for S but N in Y does not if $H = h1$, M in N in Y has V for S but M in X does not if $H = h2$. We introduce H to represent which of X and Y holds the value V. We define $diff((M,X),(N,Y),S,V,h1)$ based on $sim((M,X),(N,Y),S,V)$:

a48, a49:

```
% (M in X) and (N in Y) do not agree on V for slot S
diff((M,X),(N,Y),S,V,h1) :- compare(Q,(M,X), (N,Y)),
      hasc((M,S,V), X),
      not sim((M,X),(N,Y),S,V).
diff((M,X),(N,Y),S,V,h2) :- compare(Q,(M,X), (N,Y)),
      hasc((N,S,V), Y),
      not sim((M,X),(N,Y),S,V).
```

The answer is collected from $diff()$ predicates similarly to the $sim()$ case.

a50:

```
answer(Q, diff((M,X),(N,Y),S,V,H)) :-  compare(Q,(M,X), (N,Y))
   ,
      question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, difference),
      diff((M,X),(N,Y),S,V,H).
```

Following is the output for answering the question "What are the differences between mutated animals and mammals?", where the facts *compare*() and *sim*() are exactly matched with the case of the similarity question.

```
answer(q2,diff((body2,d1),(body2,b1),has_part,four_legs,h2))
answer(q2,diff((body2,d1),(body2,b1),has_size,big1,h2))
answer(q2,diff((d1,d1),(b1,b1),has_part,head1,h2))
answer(q2,diff((d1,d1),(b1,b1),instance_of,mammal,h2))
answer(q2,diff((d1,d1),(b1,b1),instance_of,mutated_animal,h1))
answer(q2,diff((d1,d1),(b1,b1),prototype_of,mammal,h2))
answer(q2,diff((d1,d1),(b1,b1),prototype_of,mutated_animal,h1))
compare(q2,(body2,d1),(body2,b1))
compare(q2,(d1,d1),(b1,b1))
compare(q2,(fur1,d1),(fur1,b1))
context(a1)
context(b1)
context(d1)1
diff((body2,d1),(body2,b1),has_part,four_legs,h2)
diff((body2,d1),(body2,b1),has_size,big1,h2)
diff((d1,d1),(b1,b1),has_part,head1,h2)
diff((d1,d1),(b1,b1),instance_of,mammal,h2)
diff((d1,d1),(b1,b1),instance_of,mutated_animal,h1)
diff((d1,d1),(b1,b1),prototype_of,mammal,h2)
diff((d1,d1),(b1,b1),prototype_of,mutated_animal,h1)
sim((body2,d1),(body2,b1),cloned_from,body0)
sim((body2,d1),(body2,b1),has_part,fur1)
sim((body2,d1),(body2,b1),instance_of,body)
sim((d1,d1),(b1,b1),has_part,body2)
sim((fur1,d1),(fur1,b1),instance_of,fur)
```

Following is the result of answering the question "What are the differences between prokaryotic and eukaryotic cells?" using the AURA KB. Since both eukaryotic and prokary-

otic cells are subclasses of the cell, it only used 6 classes as context for unification. It compared a total of 62 pairs of instances, finding 682 similarities and 126 differences. The first 10 differences between prokaryotic and eukaryotic cells are shown as follows.

```
answer(q2,diff((cell325,prokaryotic cell347),(cell325,
    eukaryotic_cell2547),has_part,centrosome7733,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,cytoskeleton7736,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,dna_polymerase_ii14856,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,dna_polymerase_iii14855,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,dna_polymerase10219,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,endoplasmic_reticulum7735,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,entity1168,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,entity32113,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,entity7731,h2))
answer(q2,diff((cell325,prokaryotic_cell347),(cell325,
    eukaryotic_cell2547),has_part,golgi_apparatus7734,h2))
```

3.7.3 Computing a Slot Value

This is a common type of question, and a representative example is **What is the agent of adhesion-of-water (X)?**. We can think of this question as a one-hop search of the slot value V, given an instance X and a slot name S. So, the answer is V if the relation $hasc((X,S,V),V)$ holds.

We consider another example of the question: **What chemical bond is the agent of adhesion-of-water (X)?**, in which more properties (like constraints) are posed in the question. The answer to this question is Q if the following relations hold:

```
hasc((X, S, V), X),
tc\_instance\_of(Q, Value_class),
Value_class = chemical\_bond.
```

We use *Value_class* to denote the additional property, and according to the question, the answer V must be a "chemical bond". The answer is a "hydrogen bond," which is both the agent of X and is a subclass of chemical bond.

Following is the question encoding, rules to answering, and the output of answers of "What chemical bond is the agent of adhesion-of-water?"

```
What chemical bond is the agent of adhesion of water?

Q:
question(q3).
qhas(q3, type, what).
qhas(q3, cat, value).
qhas(q3, param1, adhesion_of_water).
qhas(q3, param2, agent).
qhas(q3, param3, chemical_bond).

A:
answer(Q, hasc((X, S, V), X)) :- hasc((X, S, V), X),
      context(X),
      question(Q),
      qhas(Q, type, what),
      qhas(Q, cat, value),
      qhas(Q, param1, Class),
      qhas(q3, param2, S),
```

```
      qhas(q3, param3, Value_class),
      tc_instance_of(V, Value_class),
      has(X, prototype_of, Class).

Output:
  answer(q3,hasc((adhesion_of_water3157,agent,hydrogen_bond3449),
     adhesion_of_water3157))
  context(adhesion_of_water3157)
```

3.7.4 Check if an Assertion Is True or False

Such a question would be asking whether an assertion of the form (X,R,Y) is true. For example, "Is it true that an animal cell is a eukaryotic cell?" is asking whether **(animal_cell, is-a, eukaryotic_cell)** is true. Without telling the context, we can understand that the assertion is true within all contexts.

The rules to check if (X,R,Y) is true and answered need to be specified according to the content of the question. For this example, the rule to answer the question category "is-a" is in the following, which means that X **is-a** Y **if** X **is a descendant class of** Y. We use $tc\_subclass(X,Y)$ to tell that X is a descendant class of Y.

a51, a52, a53:

```
      tc_subclass(X,Y) :- has(X, superclass, Y).
      tc_subclass(X,Z) :- tc_subclass(X,Y), has(Y, superclass, Z).

      answer(Q,true) :- question(Q),
            qhas(Q, type, is_it_true),
            qhas(Q, cat, is_a),
            qhas(Q, param1, Class1),
            qhas(Q, param2, Class2),
            has(C1, prototype_of, Class1),
            tc_instance_of(C1, Class2).
```

In addition, we have the rule that returns $false$ when we cannot conclude that the answer is *true*.

a54:

```
answer(Q, false) :- not answer(Q,true),
      question(Q),
      qhas(Q, type, is_it_true).
```

The following is the question encoding and output of the answer to "Is it true that an animal cell is a eukaryotic cell?"

```
%  Q: Is it true that an animal cell is a eukaryotic cell?
      question(q4).
      qhas(q4, type, is_it_true).
      qhas(q4, cat, is_a).
      qhas(q4, param1, animal_cell).
      qhas(q4, param2, eukaryotic_cell).


  Output:
  answer(q4,true)
```

3.7.5 Compute Relationship Between Two Individuals

An example of this type of question is "What is the relationship between light reaction and the Calvin cycle?" or "How are light reaction and the Calvin cycle related?". In (Baral and Liang, 2012), we showed a simple answer of this question as the similarities and differences between $light-reaction$ and $Calvin-cycle$. A better answer to this question was discussed in (Baral *et al.*, 2012b), which contains both structural relation ($light-reaction$ and $Calvin-cycle$ are sub-events of *photosynthesis*) and behavioral relation ($light-reaction$ enables $Calvin-cycle$). A complete formalization of this answer is presented in another parallel work of ours about answering various types of "How" and "Why" questions.

Table 3.1: Comparison Between the Four Systems Implementing Unification

Criteria	(Clark and Porter, 2011)	(Baral and Liang, 2012)	(Chaudhri and Son, 2012)	This work
Fact type	unskolemized	skolemized	unskolemized	skolemized
Non-Destructive		Y	Y	Y
Non-monotonic		Y	Y	Y
Multiple results			Y	Y
Declarative		Y	Y	Y
Procedural	Y	Y		Y
Controllable level of details	Y		Y	Y
Principle: respect the original annotation	Y	Y	Y	Y
Principle: keep the most specific information	Y		Y	Y
Explicit unification	Y	Y		Y
Locally recursive		*		Y
Globally recursive		**		Y

3.8 Related Work

This work is a major extension of (Baral and Liang, 2012) and directly relates to (Chaudhri and Son, 2012). Both of these works are inspired by the unification procedure (Clark and Porter, 2011). In the following section, we compare this work with the three systems. Table 3.1 summarizes the differences among the systems.

3.8.1 Input Facts

First of all, both (Baral and Liang, 2012) and this work use skolemized facts, while (Clark and Porter, 2011) and (Chaudhri and Son, 2012) use unskolemized facts in AURA. (Clark and Porter, 2011) uses the unskolemized facts in KM (LISP like), as shown in section 3.2. (Chaudhri and Son, 2012) uses unskolemized facts in ASP format.

Example of unskolemized facts in KM format (from (Chaudhri and Son, 2012)):

```
Unskolemized facts in KM format:

(every Car has
   (has-engine ((a Engine called E1)))
   (has-tank ((a Tank with
      connected-to (((the has-engine
       of Self) called E1 ))))
```

Example of unskolemized facts in ASP format (from (Chaudhri and Son, 2012)):

```
Unskolemized facts in ASP format:

      class(car).
      class(engine).
      class(tank).

      % car has engine
      instance_of(_engine1(X), engine):- instance_of(X, car).
      slot(has_engine, X, _engine1(X)):- instance_of(X, car).

      % car has tank
      instance_of(_tank2(X), tank):- instance_of(X, car).
      slot(has_tank, X, _tank2(X)):- instance_of(X, car).

      %  engine and tank are connected
      slot(connected, _engine1(X), _tank2(X)):- instance_of(X, car).

      %% object_of
      object_of(_engine1(X), X) :- constant(X).
      object_of(_tank2(X), X) :- constant(X).

      codomain_type(_engine1(X), engine):- constant(X).
```

```
codomain_type(_tank2(X), tank):- constant(X).

domain_type(_engine1(X), car):- constant(X).
domain_type(_tank2(X), car):- constant(X).

%     Example facts needed for reasoning about an object s1 which is
    a suburban car.
%     constant(s1).
%     instance_of(s1, suburban).
%     subclass_of(suburban, car).
```

Skolemized version of the facts above:

```
class(car).
class(engine).
class(tank).

% car has engine
has(car1, instance_of, car).
has(car1, prototype_of, car).

has(engine1, instance_of, engine).
has(car1, has_engine, engine1).

% car has tank
has(tank1, instance_of, tank).
has(car1, has_tank, tank1).

%  engine and tank are connected
has(engine1, connected, tank1).

%     Example facts needed for reasoning about an object s1 which is
```

```
    a suburban car.
%     has(suburban, subclass, car).
%     has(s1, instance_of, suburban).
```

3.8.2 Operational Characteristics

The heuristic unification process in (Clark and Porter, 2011) was designed as a command to modify the knowledge base; thus, it has following characteristics:

1. Destructive. i.e. when an instance is unified with the other, the other is removed from the knowledge base.

2. Has only one possible result. A single default choice is not enough for many cases, such as when only one value (i.e. *animal*1 has what kind of body) can be chosen among two candidates which are equally specific (i.e. *body*1 and *body*2, both of them are instances of *body* and subsume each other)

3. Monotonic and procedural.

While the unification process in (Clark and Porter, 2011) is considered as a command to change KB, the other three can be considered as views of KB. For example, the unification of $b1$ is the view of $b1$ in the KB. All three systems are non-destructive and non-monotonic in constructing the views, i.e. the KB is not changed after unification; when KB is updated, the unification results are also changed.

The last two systems give multiple results, while (Baral and Liang, 2012) gives only one. Both (Baral and Liang, 2012) and this work describe the unification and give ASP implementation, but the algorithm can also be used for procedural implementation.

In (Baral and Liang, 2012), when $b1$ is unified and if *body*2 in $b2$ is cloned from a *body*0, *body*2 is also unified with *body*0. The result is that every instance related to $b1$ is unified. While this is perfectly fine in our toy examples, querying AURA KB returns a huge

output. Thousands of instances were unified and included in the output, but most are not closely related to the main subject. In this work, we first resolved all instances that $b1$ is cloned from and keep the other cloning untouched (such as $body2$ is cloned from $body0$). If needed, $body2$ can be unified with $body0$ in another unification round. The output of this work would be compact, human readable, and can be extended to include increasingly more information if required. The other two systems (Clark and Porter, 2011; Chaudhri and Son, 2012) can be used in a similar manner to control the level of details in the output.

3.8.3 Unification Algorithm

In this section, we compare the most important properties of the four systems; they are the main factors controlling the completeness and correctness of the unification result. For example, when $c1$ is unified and globally recursive, the system would not unify $b1$ initially to get $a1$'s information and bring to $G^*(c1)$.

Explicitly ((Chaudhri and Son, 2012) and this work) or implicitly (Clark and Porter, 2011; Baral and Liang, 2012), all four systems use two principles in section 3.4.1. While all four follow the principle of respecting the original annotation, only three follow the specificity principle. (Baral and Liang, 2012) uses the most specific instances for micro-cloning.

There are two types of cloning in KM (corresponding to two cases in Definition 21): explicit (using cloned_from slot) and inheritance. (Chaudhri and Son, 2012) only supports the last, while the others support both.

By locally recursive, we mean that the system should automatically continue unifying $body2$ (and other descendants) when it unifies $b1$. By globally recursive, we mean that the system should automatically figure out and initially unify $a1$ and then $b1$ when it needs to unify $c1$. (Baral and Liang, 2012) also considers these cases but the results do not match with common sense; it does not keep track of context and does not consider some cases of

microcloning and selecting instance by specificness. More information about this is in the next subsection. Of course, all systems can be called repeatedly on a specified list of objects to have the effect of recursiveness. For example, if we want to unify $c1$, we create a list of objects to unify, including $a1$, $body1$, $b1$, $body2$,... We then are able to call unification on an object in the list, one by one, and save the result for the next call. However, with our recursive definitions, our system will figure out what objects need to be unified and will do so on the fly.

3.8.4 More Details About the Differences Between Our Two Systems

In (Baral and Liang, 2012), we propose declarative formulation for unification and implementation in ASP. Based on the unification results, we also proposed ASP rules to answer questions that were answered in AURA system. (Baral and Liang, 2012) unifies an instance X with X's ancestor classes and from instances in which X was cloned from. Compared to (Baral and Liang, 2012), we have many improvements in this version:

1. Clear definition what is included in the an object graph. This prevents unrelated lengthy information in the results and also serves as the basis for our later formulations and proofs.

2. Improved algorithm to answer questions.

3. Included proofs for ASP implementations.

4. The context is introduced to capture the "inheritance-by-clone" of annotators. Consider the example in Figure 3.9(a), where $b1$ is annotated as "cloned-from" $a1$. It should be understood that $b1$'s body should also be inherited from $a1$'s body. This means that $body2$ should be unified with $body1$ (from $a1$) and has properties such as has-part 4-legs and has-size $big1$. However, $body2$ in $d1$ should not have those properties since $d1$ is not cloned from $a1$. In (Baral and Liang, 2012), the contexts

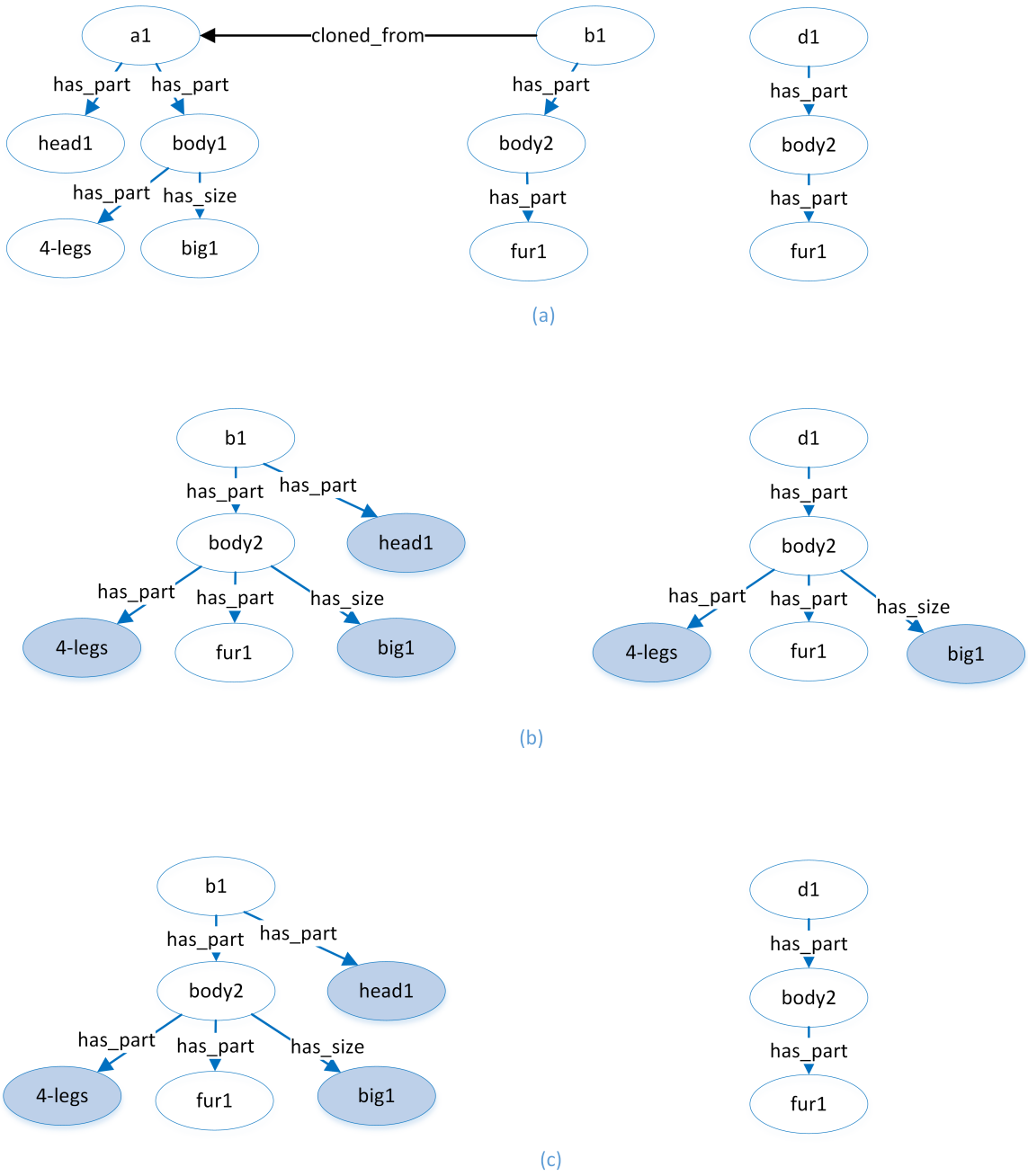

Figure 3.9: Differences between our two systems, caused by context.

(a) Object graphs of a1, b1 and d1.

(b) Unified object graphs of a1, b1 and d1 using algorithms in our previous work.

(c) Unified object graphs of a1, b1 and d1 using algorithms in this work.

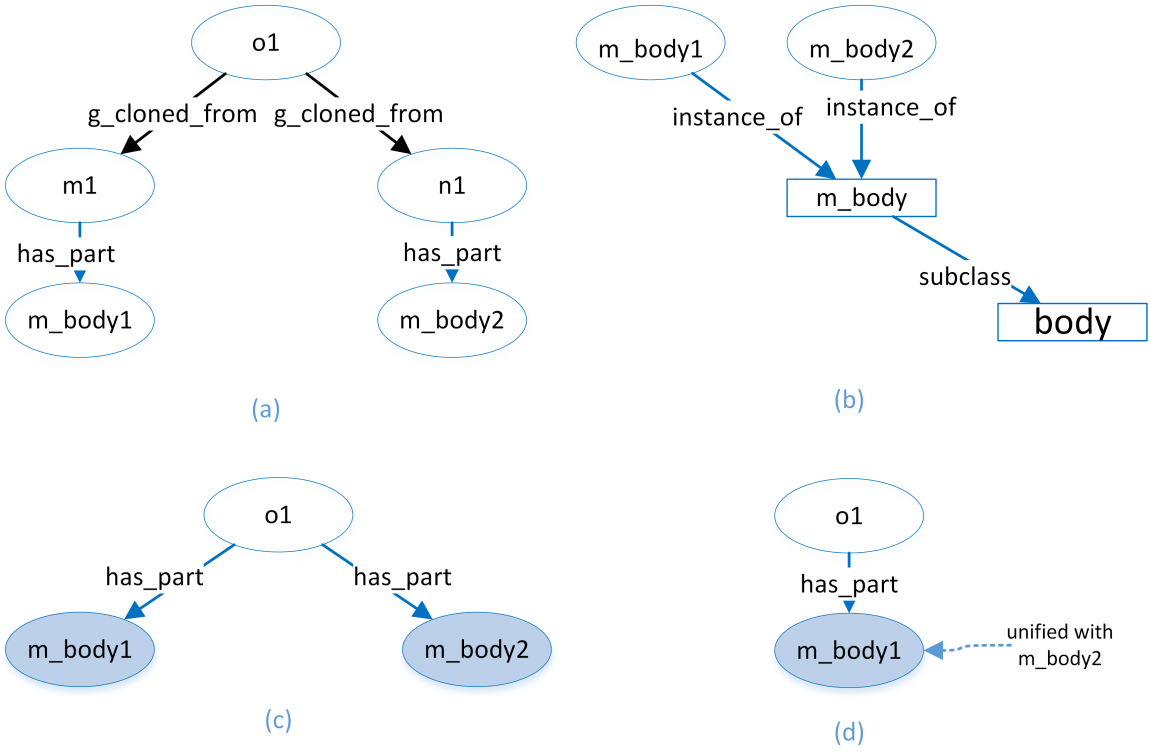

Figure 3.10: Differences between our two systems, caused by specificness equalities.
(a) Object graphs of m1, n1 and o1 and relation between m_body1, m_body2.
(b) Unified object graphs of o1 using algorithms in our previous work.
(c) Unified object graphs of o1 using algorithms in this work.

of $b1$ and $d1$ are not considered. The result is that when $b1$ is unified, new properties are added to $body2$, and $d1$ also uses those new properties. Figure 3.9(b) shows the unified object graphs of $b1$ and $d1$ using algorithms in (Baral and Liang, 2012). $body2$ in the object graph of $d1$ also have $4-legs$ and $big1$. Figure 3.9(b) shows the unified object graphs of $b1$ and $d1$ using algorithms in this work. As we expected, $body2$ in the object graph of $d1$ does not have incorrect properties, i.e. $4-legs$ and $big1$.

5. Consider equality in specificness and yield multiple outputs. Figure 3.10(a)(b) shows

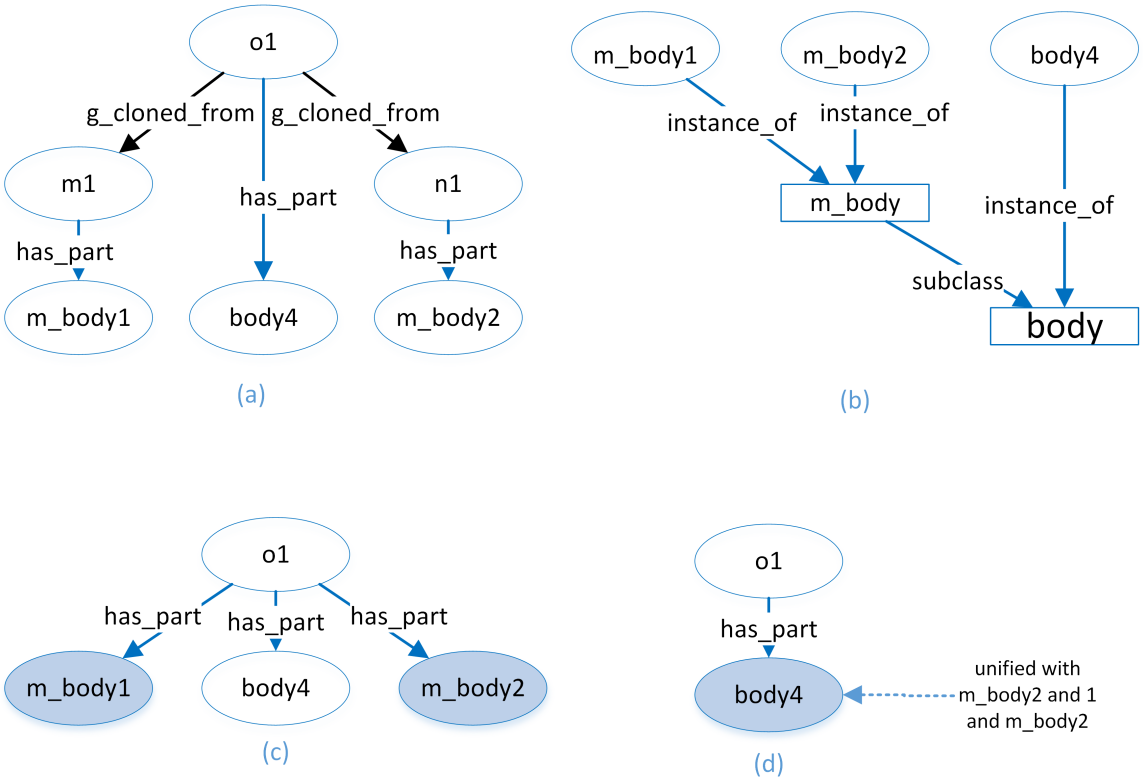

Figure 3.11: Differences between our two systems, caused by microcloning.

(a) Object graphs of m1, n1 and o1 and relation between m_body1, m_body2 and body4.

(b) Unified object graphs of o1 using algorithms in our previous work.

(c) Unified object graphs of o1 using algorithms in this work.

an example where $o1$ is g-cloned-from $m1$ and $n1$. $m1$ has m_body1 and $n1$ has m_body2. Both m_body1 and m_body2 are instances of m_body. Since m_body1 and m_body2 are equally specific (they subsume each other), our previous algorithms would give only one answer where both m_body1 and m_body2 are in the unified object graph of $o1$ (3.10(c)). This may not be correct since $o1$ should have only one body that contains properties of both m_body1 and m_body2. The algorithm in this work would return two answers, where one of m_body1 or m_body2 would be selected to be included in $o1$'s object graph. However, both answers are correct

because the selected one would be unified with the other. Figure 3.10(d) shows the case where m_body1 is selected and is unified with m_body2. It contains properties of both m_body1 and m_body2.

6. Consider more scenarios of microcloning. Figure 3.11(a)(b) shows another example of $m1$, $n1$, and $o1$. $o1$ now has part $body4$, which is an instance of $body$. Our previous algorithms would put m_body1, m_body2, and $body4$ in $o1$'s object graph, while the new one would unify both m_body1 and m_body2 with $body4$.

3.9 Conclusion and Future Work

Similar to AI planning, where there is somewhat of a mismatch in terms of research planners (mostly not HTN based) and industrial planners (mostly Hierarchical Task Network - HTN (Sacerdoti, 1974) based), there seems to be a mismatch between KR logics and the formalisms used in large knowledge bases. A large number of the latter seem to use frame-based representation while there is comparatively less theoretical research (outside of DL) on frame-based representation. Some exceptions include DLV+, FAS, and F-logic (Kifer and Lausen, 1989). In this work, we show how answer set programming can be used to encode some procedural reasoning mechanisms in frame-based systems, like "cloning" and "unification" in particular. We then show that those ASP encodings can be further enhanced with appropriate filters for making answer finding as query (pattern) driven, as in Magic Sets (Bancilhon *et al.*, 1985). Thus, we show how to answer various kinds of questions with respect to large knowledge bases constructed by domain experts with domains in a reasonable amount of time in particular. In the future, we plan to explore answering of other kinds of questions, such as "Why" and "How" with respect to such large knowledge bases.

3.10 Acknowledgment

We would like to thank Vinay Chaudhri for encouraging us to pursue research in this direction, for clarifying many of our doubts regarding AURA and frame-based knowledge representation in general, and for giving us access to an expert constructed large knowledge base.

Chapter 4

REASONING WITH CURATED KNOWLEDGE BASES: REASONING TO DERIVE MISSING INFORMATION IN EVENT-OBJECT DESCRIPTION GRAPHS LIKE KDG

Going from abstract structures to reasoning with real knowledge bases (KBs), we noticed that the KBs often have missing pieces of information, such as properties of an instance (of a class) or relations between two instances. For example, AURA does not encode that *Eukaryotic translation* is the next event of *Synthesis of RNA in eukaryote*; this may be because the two sub-events of "Protein synthesis" were encoded independently. The missing pieces make the KB and the Description Graphs constructed from it fragmented, and as a result, answers obtained with respect to them are not intuitive. Moreover, the KBs like AURA often have two or more names that refer to the same entity. To get intuitive answers, they need to be resolved and merged into a single entity. Such findings of non-identical duplicates in the KB and the act of merging them into one is referred in the literature as entity resolution (Getoor and Diehl, 2005; Brizan and Tansel, 2006).

In this chapter, we start with underspecified knowledge description graphs (UDGs) and formulate notions of reasoning with respect to these graphs to obtain certain missing information. We then present our approach of entity resolution and use it in recovering additional missing information. We give an Answer Set Programming (ASP) encoding of our formulation, and we conclude with a discussion on the uses of the above, particularly in answering "Why" and "How" questions.

4.1 Underspecified Knowledge Description Graphs

An **Underspecified Knowledge Description Graph (UDG)** is a structure that represents the facts about instances and classes of events, entities and relationships between

them. An UDG is constructed from knowledge bases such as AURA. Formal definition of the UDGs is given in the following.

Definition 31 *An UDG is a directed graph with one type of node and five types of directed edges: compositional edges, ordering edges, locational edges and participant edges. Each node represents an instance (of a class) or a class in our KBs. An UDG has the property that there are no directed cycles within any combination of compositional, locational and participant edges.*

We also used the slot names in KM (Clark *et al.*, 2004) and AURA as a guide to categorize four types of edges (Table 2.1).

Constructing the KDGs from UDGs is completing information about entity, event, and class. In this section, we discuss the missing information in the UDGs and the KDGs and how we can recover some of it through reasoning.

Coming from the KB used to construct the KDG, this missing information prevents us from reasoning on the KDG.

There are basically two types of missing information: (i) missing properties of an instance of a class or relations between two instances resulting from negligence (Section 4.2) and (ii) missing information more likely resulting from the methodology of data collection (Section 4.5, 4.6).

4.2 Event, Next Event, First Sub-event and Last Sub-event

One can directly obtain event names by looking at facts in the form of "has (E, instance of, event)" in the KB and concluding E as an event. However, for some events, such facts may be missing. In that case, we may be able get the fact from the UDG's edges and the edge constraints of the KDGs (Figure 2.1). More formally:

Definition 32 *Let E be a node in the UDG(Z). E is an event if there is*
(i) a participant edge or an ordering edge from E;

(ii) a locational edge or an ordering edge to E;

(iii) a compositional edge (of subevent or first subevent relation) from/to E; or

(iv) a path of class edges from E to the class event.

Based on Definition 32, we get that *photosynthesis* is an event because it has compositional edges (of subevent relation) to *light reaction* and the *Calvin cycle*.

Next-event, first sub-event, and last sub-event are amongst the most important properties in describing events. However, they are not always directly available in our KB. Fortunately, in many cases, we can recover them from other properties.

Definition 33 *Let E and E' be two events in the KDG(Z). Event E' is the next event of E if E enables, causes, prevents or inhibits E'.*

In other words, E' is the next event of E if there is an ordering edges from E to E'.

Definition 34 *Let S be the set of subevents of an event X in the KDG(Z). Event E in S is the first subevent of X if there exists no other event E' in S such that E is the next event of E'. Similarly, event E in S is the last subevent of X if there exists no other event E' in S such that E' is the next event of E.*

Here we assume that S was properly encoded in that there is only one chain of subevents in S. In our KB, *light reaction* and *Calvin cycle* are two sub-events of *photosynthesis*, and *light reaction* enables the *Calvin cycle*. Their orders are not defined, however. Using Definition 33 and 34, we can identify that: the *Calvin cycle* is the event following *light reaction*, *light reaction* is the first sub-event of *photosynthesis*, and the *Calvin cycle* is the final sub-event.

Two types of events: In our KB, there are two types of events: transport events and operational events. In a transport event, there is only a change in the locations; the input and output locations are different from each other while the input and output entities are the same. All other events are operational events. In an operational event, there is usually no change in its location. We differentiate the two types of events by their ancestor classes; transport events are descendants of the classes *move_through*, *move_into* and *move_out_of*.

Input, Output, Input Location, and Output Location: To reason about the KDG, we need the input and output of each event as well as the input and output locations, which are not always available. In the following, we show how to use various events' relations - such as raw-material, destination, location, and others - to create four new relations (IO relations): input, output, input-location, and output-location. After that, we propose rules to complete the KDG's IO relations.

We created the IO relations of an event based on specific relations as shown in Table 4.1. The meaning of relation "base" from AURA depends on the context. For transport events, it is for input-location; for operational events, it is for input.

Completing Missing Information of Input, Output, Input Location, and Output Location: We can obtain missing IO properties of an event from its sub-event(s). For instance, an input of the first sub-event of E is also an input of E.

Definition 35 *The input and input-location of the first subevent of E are also the input and input location of E, respectively. The output and output-location of the last subevent of E are also the output and output-location of E, respectively.*

In our KB, *photosynthesis* has two sub-events: *light reaction* and the *Calvin cycle*, the event following *light reaction*. *Sunlight* is the raw-material of *light reaction*, *sugar* is the result of the *Calvin cycle*. Using Definition 35, we have that *sunlight* is the input

Event type	IO relation type	Relation(s)
Transport event	input	object
Transport event	output	object
Transport event	input-location	base, origin
Transport event	output-location	destination
Operational event	input	object, base, raw-material
Operational event	output	result
Operational event	input-location	site
Operational event	output-location	destination [1]

Table 4.1: The IO Properties of Events and Their Corresponding Relations.

of *light reaction* as well as *photosynthesis*; *sugar* is the output of both *Calvin cycle* and *photosynthesis*.

Similarly, the output location of an operational event is often not defined in the KB, but we can use input location as the default value for output location.

Definition 36 *Let E be an event in KDG(Z), E's input location is also the output location if E's output location has not been specified.*

Figure 4.1 shows the IO properties of events in Fig 2.3. The properties in bold are the ones that were recovered using Definitions 35 and 36.

4.4 Main Class of an Instance

In our KB, one instance may belong to many classes. For example, *dna_strand19497*, the input of *Eukaryotic transcription*, is an instance of *dna_strand, dna_sequence, nucleic_acid* and *polymer* [2] . However, to reason about the equality between instances,

[2]For the sake of simplicity, in the previous figures and descriptions, we usually referenced the entities and events by their "main" class(es) and not by the instances' names although our KB and our implementation

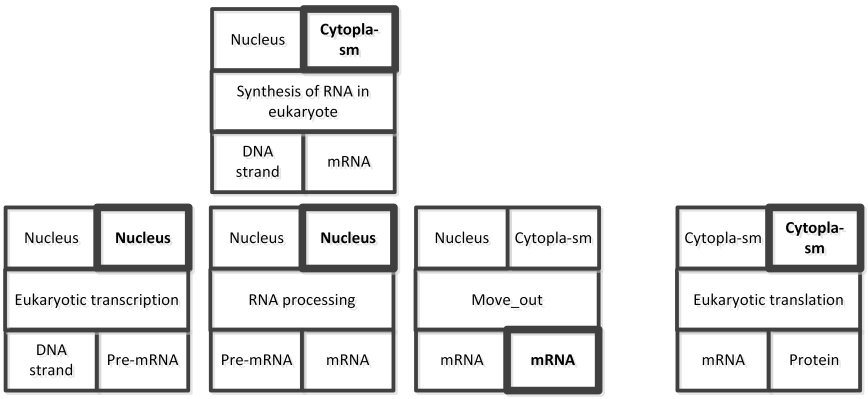

Figure 4.1: *The IO properties of events in Fig 2.3. The five blocks contain IO properties of events: Synthesis of RNA in eukaryote, Eukaryotic translation, Move_out, and Eukaryotic transcription. The middle rectangle of each block contains the event name. The top rectangles are for input and output locations; the bottom rectangles are for input and output. The properties in bold were recovered using Definitions 35 and 36*

we need the "main" class(es) - the most specific class(es) of that instance. Our formal definition of "main" class is given below.

Definition 37 *Let E be an instance in KDG(Z). ClassB is a main class of instance E if*

1. *it is a class of E and*
2. *it is not the case that there is a ClassA which is a class of E and*
   (a) *ClassB is ancestor of ClassA or*
   (b) *ClassB is a general class but ClassA is not;*

where general classes in our KB are thing, event, entity, spatial_entity, tangible_entity, and chemical_entity.

works on instances' names.

The main classes of *dna_strand19497*, according to the Definition 37, are *dna_strand* and *dna_sequence*; the other classes of *dna_strand19497* are ancestors of those two.

4.5 Entity Resolution

In the KBs such as AURA, the curation was done in many sessions and probably by many people (The same is true with respect to many other KBs; especially the ones that are developed using crowd-sourcing). The results, in many cases, (i) use two different instance names when they are probably the same instance and (ii) contain parts of some biological process encoded as independent events. For example, the input of *Eukaryotic translation* (Figure 2.3) is *mrna4642*, whereas the output of *Move_out* is *mrna22911*; *Synthesis of RNA in eukaryote* and *Eukaryotic translation* should be sub-events of "Synthesis of protein in eukaryote," but they are encoded as two separate events.

In this section, we propose methods to solve the first problem. These methods are then used to solve the second problem in the next section. In order to compare two instances in a KB, we define a match relation. Generally speaking, instance A can match with instance B if A can be safely used in a context where a term of B is expected. We defined matching relation with many confidence levels for greater flexibility in future works.

Definition 38 *Let A and B be two instances in KDG(Z). Let ClassA and ClassB be main classes of A and B respectively.*

1. *A matches with B with high confidence if one of the following is true*
   (a) *A = B (A and B are the same instance)*
   (b) *A is cloned from B (Shortcut in AURA to specify that A has all the properties of B)*
   (c) *ClassA is an ancestor of ClassB.*

2. *A matches with B with medium confidence if A and B are both cloned from an instance C.*

3. *A matches with B with low confidence if ClassA = ClassB (A and B are instances of the same main class).*

4. *A matches with B with confidence $Conf$ if all the following are true*

   (a) *A matches with C with confidence $Conf_1$*

   (b) *C matches with B with confidence $Conf_2$*

   (c) *$Conf$ is the lower confidence between $Conf_1$ and $Conf_2$.*

5. *Otherwise, A does not match with B.*

Using Definition 38, we can match *mrna4642*, the input of *Eukaryotic translation*, with *mrna22911*, the output of *Move_out*, because both have *mrna* as the main class.

While Definition 38 can match all the inputs and outputs in our aforementioned example, it is not sufficient for matching location. For example, we cannot match an instance of *cytoplasm* to an instance of *cytosol*. However, when we say *Event A occurs in cytosol*, we can understand that *Event A occurs in cytoplasm*. To overcome this shortcoming, we define the relation *Spatially match* as follows.

Definition 39 *Instance A in $KDG(Z)$ is a location instance if the class ClassA of A is a descendant of the class $spatial\_entity$.*

Definition 40 *Let A and B be two location instances in $KDG(Z)$. Let ClassA and ClassB be main classes of A and B respectively.*

1. *Location A spatially matches with location B with confidence $Conf$ if instance A matches with instance B with confidence $Conf$.*

2. *Location A spatially matches with location B with high confidence if one of the following is true:*

   (a) *B is inside A (the relation inside is encoded in our KB by slot name is_inside).*

   (b) *B is part of A (the relation "part of" is encoded in our KB by slot name part_of).*

3. *Location A spatially matches with location B with confidence $Conf$ if all the following are true:*

   (a) *A spatially matches with location C with confidence $Conf_1$.*

   (b) *C spatially matches with B with confidence $Conf_2$.*

   (c) *$Conf$ is the lowest confidence between $Conf_1$ and $Conf_2$.*

Suppose that in our KB that *cytosol234 and cytosol987 all are instances of cytosol; cytoplasm322 is an instance of cytoplasm, and cytosol987 is inside cytoplasm322*. We can then conclude: *cytosol234 and cytosol987 match with each other with low confidence, according to Definition 38.3 while cytoplasm322 spatially matches with cytosol987 with high confidence (Definition 40.2.a). Cytosol987 spatially matches with cytosol234 with low confidence (Definition 40.1), and cytoplasm322 spatially matches with cytosol234 with low confidence (Definition 40.3)* .

4.6 Finding the Possible Next Events

In this section, we demonstrate the usefulness of matching instances (Definition 38 and 40) in finding the next possible event(s) of a given event. While in simple cases (Section 4.2) we can find the next event E' of an event E by using Definition 33, there are still cases where no ordering edges from E to E' exists. For examples, *Alteration of mrna ends* and *RNA splicing* are two sub-events of *RNA processing*, but no other relation between them was defined. However, they all occur in *nucleus16421*, and *Alteration of mrna ends*'s output,

pre_mrna7690, matches with *RNA splicing*'s input, *rna8697*. This information suggests that *RNA splicing* is *Alteration of mrna ends*'s next event.

Following this intuition, our approach for finding the next event is that E' is the subsequent event of E if the output of E matches the input of E', and output location of E matches the input location of E'. In the example in Figure 2.3, this assumption holds in all three events: *Eukaryotic transcription*, *RNA processing* and *Move_out*, all of which are already defined in our KB as consequent events. This assumption also suggests that *Eukaryotic translation* can be the following event of either *Synthesis of RNA in eukaryote* or *Move_out*. Armed with Definition 38 and 40, we define the following join relation.

Definition 41 *Let A and B be two events in KDG(Z). Event A joins to event B if all of the following conditions are true:*

1. *The output of A matches with the input of B or vice versa.*
2. *The output location of A spatially matches with the input location of B or vice versa.*

Applying this definition, *Alteration of mrna ends* joins to *RNA splicing*, *Eukaryotic transcription* joins to *RNA processing*, *RNA processing* joins to *Move_out*, and both *Synthesis of RNA in eukaryote* and *Move_out* join from *Eukaryotic translation*. Since we want *Eukaryotic translation* to be a possible next event of *Synthesis of RNA in eukaryote* instead of its sub-event *Move_out*, we define the possible next event as follows.

Definition 42 *Let A and B be two events in KDG(Z) where A joins to B. B is a possible next event of A if none of the following conditions is true:*

1. *A joins to AncestorB where AncestorB is the ancestor event of B (in other words, there is a non-empty path of subevent relation from AncestorB to B).*
2. *Ancestor event AncestorA of A joins to B.*

3. *A is an ancestor event of B.*

4. *B is an ancestor event of A.*

5. *A and B have the same ancestor event.*

In our example, condition 42.2 gives us that *Eukaryotic translation* is not the possible next event of *Move_out* while 42 concludes that *Eukaryotic translation* is the possible next event of *Synthesis of RNA in eukaryote*. We assume that an event and its sub-events are put in our KB as a whole, so the *next_event* relations between them are well defined. Thus, conditions 42.3-5 take those relations out of consideration.

When we have a path of possible next events, we can create an event *SE*, which is the super event of all events in the path, and add suitable *next_event* or *subevent* relations. This new event would link the events that were mistakenly encoded as independent events (that we mentioned earlier).

4.7 ASP Encodings

4.7.1 Encoding the Types of Edges

Similar to the encoding of types of edges in 5.1.1.

4.7.2 Recovering Event and Entity Information

Finding Next Events, First Sub-events and Last Sub-events: Rules ee1-ee2 find the next events (Definition 33), and rules ee3-ee6 find the first and last sub-events (Definition 34).

```
ee1: predicates(ordering_edge, enables; causes; prevents; inhibits).
ee2: has(E1, next_event, E2) :- has(E1, Predicate, E2), predicates(
   ordering_edge, Predicate).
ee3: not_fse(Z, E) :- has(Z, subevent, E), has(Z, subevent, E2), E2
   != E, has(E2, next_event, E).
```

```
ee4: not_lse(Z, E) :- has(Z, subevent, E), has(Z, subevent, E2), E2
   != E, has(E, next_event, E2).
ee5: has(Z, first_subevent, E) :- has(Z, subevent, E), not not_fse(Z
   , E).
ee6: has(Z, last_subevent, E) :- has(Z, subevent, E), not not_lse(Z,
    E).
```

4.7.3 Encoding Transport Events and Operational Events

$t\_event(E)$ or $o\_event(E)$ is used to indicate a transport event or an operational event, respectively.

```
ev1: predicates(t_event, move_through; move_into; move_out_of).
ev2: t_event(E) :- has(E, instance_of, Transport_class), predicates(
   t_event, Transport_class), event(E).
ev3: o_event(E) :- event(E), not t_event(E).
```

4.7.4 Encoding the Inputs and Outputs of Operational Events

We denote the input/output/input location/output location of an event with *input*, *output*, *input_loc* and *output_loc* respectively. Rules i1-i6 get the IOs of operational events. IOs of transport events are encoded similarly (rules i7-i11).

```
i1: input(E, A) :- has(E, object, A), o_event(E).
i2: input(E, A) :- has(E, base, A), o_event(E).
i3: input(E, A) :- has(E, raw_material, A),o_event(E).
i4: output(E, A) :- has(E, result, A), o_event(E).
i5: input_location(E, A) :- has(E, site, A), o_event(E).
i6: output_location(E, A) :- has(E, destination, A), o_event(E).

i7: input(E, A) :- has(E, object, A), t_event(E).
i8: output(E, A) :- has(E, object, A), t_event(E).
i9: input_location(E, A) :- has(E, base, A), t_event(E).
i10: input_location(E, A) :- has(E, origin, A), t_event(E).
```

```
i11: output_location(E, A) :- has(E, destination, A), t_event(E).
```

Getting the Missing Inputs and Outputs: Rule i12 gets the input of an event from its first sub-event (Definition 35). Other rules are used to get the input location, output, and output location. Rule i16 gets the default output location of an event(Definition 36).

```
i12: input(E, A) :- input(SE, A), has(E, first_subevent, SE).
i13: input_location(E, A) :- input_location(SE, A), has(E,
   first_subevent,
SE).
i14: output(E, A) :- output(SE, A), has(E, last_subevent, SE).
i15: output_location(E, A) :- output_location(SE, A), has(E,
   last_subevent
, SE).

i16: has(E, output_location, A) :- not has(E, output_location, A2),
   has(E, input_location, A),  entity(A2), event(E), A2 != A.
```

Encoding the Main Class(es) of an Instance: *ClassA* is a main class of instance *A* if *ClassA* is one of *A*'s classes and we do not have $not\_main\_class(A, ClassA)$ (which mean *ClassA* is not the main class of *A*).

```
m1: general_class(thing; event; entity; spatial_entity;
   tangible_entity; chemical_entity).
m2: not_main_class(A, ClassB) :- has(A, instance_of, ClassA), has(A,
    instance_of, ClassB), has(ClassA, ancestorclass, ClassB).
m3: not_main_class(A, ClassB) :- has(A, instance_of, ClassA)), has(A
   , instance_of, ClassB), general_class(ClassB), not general_class(
   ClassA).
m4: main_class(A, ClassA) :- has(A, instance_of, ClassA), not
   not_main_class(A, ClassA).
```

Encoding Instance Matching: We use predicate $match\_with(A,B,Confidence)$ to represent *match with* relations (Definition 38) from instance A to B; $Confidence$ can be either *low, medium,* or *high*. Rule ma1 encodes the sub-case 38.1.a of Definition 38. The last rule is for Definition 38.4, matching A to B transitively through C. $lowest\_confidence(Conf1,Conf2,Conf)$ means $Conf$ is the lowest confidence in $Conf1$ and $Conf2$ (Rules lc1-lc7). Rules for other cases of Definition 38 are ma2-ma5.

```
% Define confidence levels
lc1: confidence(high;medium;low).
lc2: lower_confidence(high,medium).
lc3: lower_confidence(medium, low).
lc4: lower_confidence(high, low).
% Z is the highest confidence among X and Y.
lc5: lowest_confidence(X, X, X) :- confidence(X).
lc6: lowest_confidence(X, Y, Y) :- lower_confidence(X, Y).
lc7: lowest_confidence(X, Y, X) :- lower_confidence(Y, X).

ma1:  match_with(A,B,high) :- main_class(A,ClassA), main_class(B,
   ClassB), A==B.
ma2: match_to(A, B, high) :- main_class(A, ClassA), main_class(B,
   ClassB), A == B.
ma3: match_to(A, B, high) :- main_class(A, ClassA), main_class(B,
   ClassA), has(A, cloned_from, B).
ma4: match_to(A, B, medium) :- main_class(A, ClassA), main_class(B,
   ClassA), main_class(C, ClassC), not match_to(A, B, high), A != B,
    A != C, B != C, has(A, cloned_from, C), has(B, cloned_from, B).
ma5: match_to(A, B, low) :- main_class(A, ClassA), main_class(B,
   ClassA), A != B, not match_to(A, B, high), not match_to(A, B,
   medium).
ma6:  match_with(A,B,Conf) :- match_with(A,C,Conf1), match_with(C,B,
   Conf2), A!=B, A!=C, B!=C, lowest_confidence(Conf1,Conf2,Conf).
```

Encoding locational instance matching Predicate $match\_location(A, B, Confidence)$ represents *spatially match to* relation (Definition 40) from instance A to B. Again, $Confidence$ can be either *low, medium,* or *high*. The first rule encodes Definition 39; the sequent rules encode Definition 40. The second rule involves the case where instance A matches with instance B; the next two rules, when B is inside or part of A; and the last rule, when A spatially matches with B through C.

```
sma1: location(A) :- has(A, instance_of, ClassA), has(ClassA,
    ancestorclass, spatial_entity).
sma2: match_location(A, B, Confidence) :- location(A), location(B),
    match_with(A, B, Confidence).
sma3: match_location(A, B, high) :- location(A), location(B), has(B,
     is_inside, A).
sma4: match_location(A, B, high) :- location(A), location(B),has(A,
    part_of, B).
sma5: match_location(A, B, Conf) :- match_location(A, C, Conf1),
    match_location(C, B, Conf2), A != B, A != C, B != C,
    lowest_confidence(Conf1, Conf2, Conf).
```

Encoding join relation Join relation (Definition 41) is encoded in the following: $\_match\_with(A, B)$ is true if A *match to* B or vice versa at any confidence level. $\_match\_location(A, B)$ is defined for similar purpose. $\_join(A, B)$ means A joins to B, which is true when we can match (regardless of direction) output of A to input of B and output location of A to input location of B.

```
j1: _match_with(A, B) :- match_with(A, B, Confidence).
j2: _match_with(B, A) :- match_with(A, B, Confidence).
j3: _match_location(A, B) :- match_location(A, B, Confidence).
j4: _match_location(B, A) :- match_location(A, B, Confidence).
j5: _join(A, B) :- output(A, OutputA), output_loc(A, OutputLocationA
    ),
```

```
j6: input(B, InputB), input_location(B, InputLocationB), A != B,
    _match_with(OutputA, InputB), _match_location(OutputLocationA,
    InputLocationB).
```

Encoding Possible-next-event Relation: In this section, we show how Definition 42 is encoded. We use $has(A, tc\_subevent, B)$ to represent transitive closure of *sub event* relation between A and B (encoded by $has(A, subevent, B)$), which is defined in the standard way (rules tcsub1-tcsub2). We also use $\_join(A, B)$ to encode that A joins to B according to Definition 41.

```
tcsub1: has(A, tc_subevent, B) :- has(A, subevent, B).
tcsub2: has(A, tc_subevent, C) :- has(A, tc_subevent, B), has(B,
    subevent, C).

% Ancestor event of A joins to B
n1: _notNextEvent(A, B) :- _join(A, SuperB), _join(A, B), has(SuperB
    , tc_subevent, B).
% A joins to to ancestor event of B
n2: _notNextEvent(A, B) :- _join(SuperA, B), _join(A, B), has(
    SuperA, tc_subevent, A).

% An event joins to its ancestor event
n3: _notNextEvent(E, A) :- _join(E, A), has(E, tc_subevent, A).
n4: _notNextEvent(A, E) :- _join(A, E), has(E, tc_subevent, A).

% Two sub events of one ancestor
n5: _notNextEvent(A, B) :- _join(A, B), has(E, tc_subevent, A), has(
    E, tc_subevent, B).

n6: possible_next_event(A,B) :- _join(A, B), not _notNextEvent(A, B)
    .
```

4.7.5 Correctness of the ASP rules

Definition 43 *The ASP program Π_C is the answer set program consisting of the following rules:*

- *ee1 to ee6 for next events, first subevents and last events,*
- *ev1 to ev3 for two types of events,*
- *i1 to i16 for inputs, outputs of events,*
- *m1 to m4 for main class(es),*
- *lc1 to lc7 for the lowest confidence,*
- *ma1 to ma6 for match relation,*
- *sma1 to sma5 for spatially match relation,*
- *tcsub1 to tcsub2 for transitive closure of subevents,*
- *j1 to j6 for joined events, and*
- *n1 to n6 for possible next events.*

Proposition 5 *E is the last subevent of X in $KDG(z)$ iff*

$$\Pi_G(z) \cup \Pi_{path} \cup \Pi_C \models has(z, last\_subevent, E)$$

where $\Pi_G(z)$ and Π_{path} were defined in Definition 44 and 46.

Proof 11 (Proposition 5) ***Forward direction:*** *If E is the last subevent of X in $KDG(z)$, all answer sets of $\Pi_G(z) \cup \Pi_{path} \cup \Pi_C$ contain $has(z, last\_subevent, E))$.*

1. *Since E is an event of X, $has(z, subevent, E)$ must be true in all answer sets.*

2. *If the body of rule $ee4$ is satisfied, there will be another event $E2$ which is a next event of E. E is not the last subevent of X : contradiction.*

3. *Hence, the body of rule $ee4$ is not satisfied. $not\_lse(X, E)$ is not true in any answer set.*

4. *Rule $ee6$, then, makes $has(z, last\_subevent, E)$ true in all answer set of $\Pi_G(z) \cup \Pi_{path} \cup \Pi_C$.*

Backward direction: *If an answer set ANS of $\Pi_G(z) \cup \Pi_{path} \cup \Pi_C$ contains $has(z, last\_subevent, E)$, E is the last subevent of X.*

1. *$has(z, last\_subevent, E)$ must be introduced by rule $ee6$; ANS contains $has(z, subevent, E))$ and does not contain $not\_lse(X, e)$. E is a subevent of X.*

2. *If E is not the last subevent of X, by definition, there must be another event $E2$ so that $E2$ is also a subevent of X and $E2$ is the next event of E. In that case, the body of rule $ee4$ must be satisfied and ANS would contain $not\_lse(X, e)$.*

3. *Hence, E is the last subevent of X.*

Proposition 6 *A is a main class of E in $KDG(z)$ iff*

$$\Pi_G(z) \cup \Pi_{path} \cup \Pi_C \models main\_class(E, A)$$

Proof 12 (Proposition 6) ***Forward direction:*** *If A is a main class of E in $KDG(z)$, all answer sets of $\Pi_G(z) \cup \Pi_{path} \cup \Pi_C$ contain $main\_class(E, A)$.*

1. *Since A is a class of E, $has(E, instance, A))$ must be true in all answer sets.*

2. *If the body of rule m2 (or m3) is satisfied, there will be another class A2 of E which is more specific than A. For example, A is a ancestor class of A2. By definition, A is not the main class of E : contradiction.*

3. *Hence, the body of rule m2 (or m3) is not satisfied. not_main_class(E,A) is not true in any answer set.*

4. *Rule m4, then, makes main_class(E,A) true in all answer set of* $\Pi_G(z) \cup \Pi_{path} \cup \Pi_C$

Backward direction: *If an answer set ANS of* $\Pi_G(z) \cup \Pi_{path} \cup \Pi_C$ *contains main_class(E,A), A is a main class of E.*

1. *main_class(E,A) must be introduced by rule m4; ANS contains has(E,instance,A) and does not contain not_main_class(E,A). E is an instance of class A.*

2. *If A is not a main class of E, there will be another class A2 of E which is more specific than A. For example, A is a ancestor class of A2 and E is also an instance of A2. In that case, the body of rule m2 (or m3) must be satisfied and ANS would contain not_main_class(E,A): contradiction.*

3. *Hence, A is a main class of E.*

4.8 Conclusion and Discussions

While using the frame based knowledge base AURA to formulate the answers to "How" and "Why" questions, we noticed missing information in the KB that make our answer less intuitive. These flaws exist not only in AURA but also in large curated KBs, often the KBs created by multiple people, sometimes even through crowd-sourcing. This often leads to some information being inexplicably stated, even though the knowledge base contains clues to derive that information. Using the examples that we observed in AURA, we developed

several general formulations regarding missing knowledge about events. Being able to obtain missing information and to enrich the original KDGs, one can obtain more accurate and intuitive answers to the deep reasoning questions.

One of our formulations was about entity resolution where we resolve multiple entities that may have different names but refer to the same entity. Our method is different from other methods in the literature of (Getoor and Diehl, 2005; Brizan and Tansel, 2006). Since each entity resolution method heavily relies on the properties of the database it is working on and no other system we know of is about AURA or similar event centered knowledge bases, we were unable to directly compare our method with those of the others. Our approach to use rules (albeit ASP rules) in deriving missing information is analogous to the use of rules in data cleaning and in improving data quality (Herzog *et al.*, 2007; Rahm and Do, 2000; Fan *et al.*, 2009). However, those works do not focus on the issues discussed in this work.

Chapter 5

ANSWERING DEEP QUESTIONS USING ANSWER SET PROGRAMMING - TOWARD AN OVERALL SYSTEM

In this chapter, we present the encoding of Knowledge Description Graphs in Answer set programming (ASP), and the encoding of the structures answering "How" and "Why" questions in previous sections.

5.1 ASP Encodings

5.1.1 Encoding the Common Components

In this subsection, we present the ASP encoding of the common components used to answer various question types.

Encoding the KDG

The $KDG(z)$ is encoded by the ASP program $\Pi_G(z)$ defined below.

Definition 44 *The ASP program $\Pi_G(z)$ is the answer set program consisting of the facts that are generated from all the nodes and edges of $KDG(z)$ in the following way:*

1. *Generate "$context(z)$".*
2. *For each node N, generate "$has(N, instance\_of, entity)$" if N is an event node, "$has(N, instance\_of, entity)$ if N is an entity node, and $class(N)$ if N is a class node.*
3. *For each edge of relation R from $E1$ to $E2$, generate 'has$(E1, R, E2)$".*

Encoding the Questions

For efficiency, we include the predicate "question" in many of our rules. This focuses the answer set computation on only those KDGs and KDG properties imperative to answering specific questions asked.

We use the same template for all of the questions. Each question has a *QID*, *Type*, *Category*, two *Parameters*, and optionally, a *Scope*.

```
q1: question(QID).
q2: has(QID, type, Type).
q3: has(QID, category, Category).
q4: has(QID, param1, XClass).
q5: has(QID, param2, YClass).
q6: has(QID, scope, ScopeClass).
```

Illustrated below are the encodings of the question "How does photosynthesis work?".

```
question(q).
has(q, type, how).
has(q, category, work).
has(q, param1, photosynthesis).
```

Definition 45 *The ASP program $\Pi_Q(q)$ is the answer set program consisting of the rules q1 to q6 to encode the question q.*

Encoding the Entities and Events

The following rules define events, entities, and things. $class(X)$ means that X is a class.

```
t1: event(X)      :- has(X,instance_of,event).
t2: entity(X)     :- has(X,instance_of,entity).
t3: thing(X)      :- entity(X).
t4: thing(X)      :- event(X).
t5: thing(X)      :- class(X).
```

Encoding the Types of Edges and Types of Paths

We use $doconnects(X,Y)$ to denote the ordering edge from X to Y. Predicate *ordering_edge* is used to define the slot names corresponding to the ordering edges. Other types of edges are defined similarly.

```
e1: compositional_edge(subevent; first_subevent).
e2: ordering_edge(next_event; enables; causes; prevents; inhibits).
e3: participant_edge(agent; beneficiary; destination; donor;
   experiencer; instrument; object; origin; path; raw_material;
   recipient; result; site; substrate; toward).
e4: locational_edge(base_of; site_of; location_of).
e5: compositional_edge(has_part; has_basic_structural_unit;
   has_region).
e6: class_edge(instance_of; super_class).

e7: doconnects(X, Y)    :-    has(X, S, Y), ordering_edge(S), event(
   X), event(Y).
e8: dcconnects(X, Y)    :-    has(X, S, Y), compositional_edge(S),
   event(X), event(Y).
e9: dcconnects(X, Y)    :-    has(X, S, Y), compositional_edge(S),
   entity(X), entity(Y).
e10: dpconnects(X, Y)   :-    has(X, S, Y), participant_edge(S),
   event(X), entity(Y).
e11: dlconnects(X, Y)   :-    has(X, S, Y), locational_edge(S),
   entity(X), event(Y).
e12: dclconnectes(X, Y) :-    has(X, S, Y), class_edge(S), entity(X)
   , class(Y).
e13: dclconnectes(X, Y) :-    has(X, S, Y), class_edge(S), event(X),
    class(Y).
e14: dclconnectes(X, Y) :-    has(X, S, Y), class_edge(S), class(X),
    class(Y).
```

We define $dcpath\_connects(X,Y)$ to be true if there is a compositional edge, participant edge, or locational edge from X to Y (rules c1-c3). $cpath\_connects$ denotes the transitive closures of $dcpath\_connects(X,Y)$; $oconnects$ denotes the transitive closures of $do\_connects(X,Y)$. $cpath\_connects2(X,Y)$ is a shortcut for $cpath\_connects(X,Y)$ or $X = Y$.

```
c1:   dcpath_connects(X, Y)   :- dcconnects(X, Y).
c2:   dcpath_connects(X, Y)   :- dpconnects(X, Y).
c3:   dcpath_connects(X, Y)   :- dlconnects(X, Y).
c4:   cpath_connects(X, Y)    :- dcpath_connects(X, Y).
c5:   cpath_connects(X, Y)    :- cpath_connects(X, K),
   dcpath_connects(K, Y), K != X, K!= Y.
c6: cpath_connects2(X, X)     :- thing(X).
c7: cpath_connects2(X, Y)     :- cpath_connects(X, Y).
c8: oconnects(X, Y)     :- doconnects(X, Y).
c9: oconnects(X, Y)     :- oconnects(X, K), doconnects(K, Y), K != X
   , K!= Y.
```

5.1.2 Answering Question "How Does X Work?"

Getting $(Z,X)\_KDG$ From the Question

We represent the answer to "How does X work?" by $zx\_kdg(Q,Z,X)$, where Q is the question ID. The root of our context, $KDG(Z)$, is represented by $context(Z)$.

```
z1: zx_kdg(Q,Z,X) :- question(Q), has(Q,type,how), has(Q,category,
   work), has(Q,param1,XClass), has(X, instance_of,XClass), context(
   Z), cpath_connects(Z,X).
```

Getting All Nodes and Edges in $(Z,X)\_KDG$

The next step of our encoding is to put all of the nodes and edges in one of the $(Z,X)\_KDG$ into the final answer. Numstep in the rule p0 below is a constant to limit the maximum

length of the cpath we want to compute. In p1-p2 by $selected\_path(A,Z,X,N)$, we mean that X is a node in the selected cpath from A to Z at distance N from A in that cpath. If a node B has already been selected, p2 selects only one node C amongst all of the nodes that directly connect from B (through cpath) and also connect to Z (or be Z itself). Rules p3-p4 remove all answer sets that do not contain the entire path from A to Z. Rule a1 gets all the nodes on the selected path from Z to X, while a2 gets all the nodes in $KDG(X)$. The last rule is to get all of edges of $(Z,X)\_KDG$.

```
p0: step(0..numstep).
p1: selected_path1(A, Z, A, 0) :- zx_kdg(Q,A,Z).
p2: 1 { selected_path1(A, Z, C, T+1) : cpath_connects2(C, Z) :
    dcpath_connects(B, C) } 1 :- selected_path1(A, Z, B, T), B != Z,
    step(T), zx_kdg(Q,A,Z).
p3: completed_path1(zx_kdg(Q,A,Z)) :- zx_kdg(Q,A,Z), selected_path1(
    A, Z, Z, T),  step(T).
p4: :- zx_kdg(Q,A,Z), not completed_path1(zx_kdg(Q,A,Z)).

a1: ans(Q,zx_kdg(Q,Z,X),node,N) :- zx_kdg(Q,Z,X), selected_path1(Z,X
    ,N,T), step(T).
a2: ans(Q,zx_kdg(Q,Z,X),node,N) :- zx_kdg(Q,Z,X), cpath_connects(X,N
    ).

% Adding all the edges in the KDG between two nodes of structure G
    to the answer.
a3: ans(Q,G,edge,has(A,Predicate,B)) :- has(A,Predicate,B), ans(Q,G,
    node,A), ans(Q,G,node,B).
```

Correctness of the ASP Rules

Definition 46 *The ASP program Π_{path} is the answer set program consisting of the rules:*

- *t1 to t5 for events and entities,*

- *e1 to e14 for edges, and*
- *c1 to c9 for paths.*

Corollary 4 *Let x and y be two nodes in the KDG(z). There exists an ordering edge (or respectively compositional, participant, locational, class edge) from x to y iff* $\Pi_G(z) \cup \Pi_{path} \models doconnects(x,y)$ *(or* $dcconnects(x,y)$, $dpconnects(x,y)$, $dlconnects(x,y)$, $dclconnects(x,y)$ *respectively).*

Corollary 5 *Let x and y be two nodes in the KDG(z). There exists a cpath edge from x to y iff*

$$\Pi_G(z) \cup \Pi_{path} \models dcpath\_connects(x,y).$$

Proposition 7 *Let x and y be two nodes in the KDG(z). There exists a cpath from x to y iff*

$$\Pi_G(z) \cup \Pi_{path} \models cpath\_connects(x,y).$$

Definition 47 *The ASP program* $\Pi^1_{answer}(numstep)$ *is the answer set program consisting of the rules:*

- *z1 for defining* $(Z,X)\_KDG$
- *p0 to p4 for selecting a cpath*
- *a1 to a3 for selecting nodes and edges for the answer set*

where $numstep$ *is the constant in rule p0.*

Definition 48 *The ASP program* $\Pi^1(q,z,numstep)$ *is the answer set program:*

$$\Pi^1(q,z,numstep) = \Pi_G(z) \cup \Pi_{path} \cup \Pi_Q(q) \cup \Pi^1_{answer}(numstep)$$

Proposition 8 *Let $q1$ be the question "How does xc work?". Let x be a node in the $KDG(z)$ so that xc is x's class. For each (z,x)_KDG K, there exists an answer set of $\Pi^1(q1,z,numstep)$ containing:*

$$ans(q1,zx\_kdg(q1,z,x),node,N)$$

and

$$ans(q1,zx\_kdg(q1,z,x),edge,has(I,Predicate,J))$$

where $numstep$ is equal to or greater than the length of the cpath from z to x in K; N is any node of K, and $(I,Predicate,J)$ is any edge of K.

Proposition 9 *Let $q1$ be the question "How does xc work?". Let x be a node in the $KDG(z)$ so that xc is x's class. Suppose there exists at least one (z,x)_KDG where the length of cpath from z to x is less than or equal numstep. If A is an answer set of $\Pi^1(q1,z,numstep)$, let $V = \{n : ans(q1,zx\_kdg(q1,z,x),node,n) \in A\}$ and $E = \{(i,Predicate,j) : ans(q1,zx\_kdg(q1,z,x),edge,has(i,Predicate,j)) \in A\}$. There exists a (z,x)_KDG K so that V is K's set of nodes and E is K's set of edges.*

5.1.3 Answering Question "How Does X Produce Y?"

We used the same question template as previous question to encode "How does X produce Y?". For example, illustrated below are the encodings of question "How does a plant produce sugar?".

```
question(q).
has(q, type, how).
has(q, category, produce).
has(q, param1, plant).
has(q, param2, sugar).
```

Encoding Output of an Event

We define that Y is the output of event X if it is produced by X or by a sub-event of X. This definition is encoded in rules o1-o2 below.

```
o1: output(E,A) :- has(E,result,A), event(E).
o2: output(E,A) :- output(SE,A), has(E,subevent,SE).
```

Getting HI_P^X

Below, we present the encoding of HI_P^X, the highest node in the $KDG(X)$ that has property P. In the first rule, the property "produce Y" of the question "How does X produce Y?" is denoted as $property(produce(YClass))$, where $YClass$ is the class of Y. The next rule defines when an event node HI in $KDG(X)$ has the property "produce Y". h3-h4 encode HI_P^X using default negation. Node X is the highest node that has property P if there is no ancestor X' of X that also has property P.

```
h1: property(produce(YClass)) :- question(Q), has(Q,type,how), has(Q
    ,category,produce), has(Q,param1,XClass), has(Q,param2,YClass),
    has(X,instance_of, XClass).
h2: has_prop(X,produce(YClass)) :- property(produce(YClass)), output
    (X,Y), has(Y,instance_of,YClass).
h3: _not_hn(X,HI,Prop) :- has_prop(HI,Prop), has_prop(AncestorHI,
    Prop), cpath_connects(AncestorHI,HI), AncestorHI!=HI,
    cpath_connects2(X, AncestorHI).
h4: _hn(X,HI,Prop) :- has_prop(HI,Prop), not _not_hn(X,HI, Prop),
    cpath_connects2(X, HI).
```

Similarly to answering the previous question type, we define $zx\_kdg(Q,X,HI)$ & $zx\_kdg(Q,Z,X)$ in z2 & z3; they correspond to $(X,HI)\_KDG$ and $(Z,X)\_KDG$.

```
z2: zx_kdg(Q,X,HI) :- question(Q), has(Q,type,how), has(Q,category,
    produce), has(Q,param1,XClass), has(Q,param2,YClass), has(X,
```

```
    instance_of,XClass), context(Z), cpath_connects2(Z,X), _hn(X,HI,
    produce(YClass)), X!=HI.
z3: zx_kdg(Q,Z,X) :- question(Q), has(Q,type,how), has(Q,category,
    produce), has(Q,param1,XClass), has(Q,param2,YClass), has(X,
    instance_of,XClass), context(Z), cpath_connects2(Z,X), _hn(X,HI,
    produce(YClass)), X==HI.
```

Definition 49 *The ASP program Π_{output} is the answer set program consisting of the rules o1 & o2*

Definition 50 *The ASP program $\Pi^2_{answer}(numstep)$ is the answer set program consisting of the following rules:*

- *z2 to z3 for defining $(Z,X)\_KDG$*
- *p0 to p4 for selecting a cpath*
- *a1 to a3 for selecting nodes and edges for the answer set*

where numstep is the constant in rule p0.

Definition 51 *The ASP program $\Pi_{highest\_node}$ is the answer set program consisting of the rules h1 to h5.*

Definition 52 *The ASP program $\Pi^2(q,z,numstep)$ is the answer set program:*

$$\begin{aligned}\Pi^2(q,z,numstep) =& \Pi_G(z) \cup \Pi_{path} \cup \Pi_{output} \cup \Pi_Q(q) \\ & \cup \Pi_{highest\_node} \cup \Pi^2_{answer}(numstep)\end{aligned}$$

Compared to $\Pi^1(q,z,numstep)$, $\Pi^2(q,z,numstep)$ has different rules for $(Z,X)\_KDG$ in $\Pi^2_{answer}(numstep)$ and additional rules in Π_{output} and $\Pi_{highest\_node}$.

Corollary 6 *Let q2 be the question "How does xc produce yc?". Let x be a node in the KDG(z). x has property "produce yc" iff*

$$\Pi^2(q2, z, numstep) \models has\_prop(x, produce(yc))$$

Corollary 7 *Let q2 be the question "How does xc produce yc?". Let x be a node in the KDG(z). he is the highest node in the KDG(x) that has property "produce yc" iff*

$$\Pi^2(q2, z, numstep) \models \_hn(x, he, produce(yc)).$$

Proposition 10 *Let q2 be the question "How does xc produce yc?", x be the highest node in the KDG(z) that has the property "produce yc," and xc be x's class. For each (z,x)_KDG K, there exists an answer set of $\Pi^2(q2, z, numstep)$ containing:*

$$ans(q2, zx\_kdg(q1, z, x), node, N)$$

and

$$ans(q2, zx\_kdg(q1, z, x), edge, has(I, Predicate, J))$$

where numstep is equal to or greater than the length of the cpath from z to x in K. N is any node of K, and $(I, Predicate, J)$ is any edge of K.

Proposition 11 *Let q2 be the question "How does xc produce yc?" and x be the highest node in the KDG(z) that has property "produce yc" and xc is x's class. If A is an answer set of $\Pi^2(q2, z, numstep)$, let $V = \{n : ans(q2, zx\_kdg(q2, z, x), node, n) \in A\}$ and $E = \{(i, Predicate, j) : ans(q2, zx\_kdg(q2, z, x), edge, has(i, Predicate, j)) \in A\}$. There exists an (z,x)_KDG K so that V is K's set of nodes and E is K's set of edges.*

5.1.4 Answering Question "How Is X Related to Y?"

Illustrated below are the encodings of the question "How is sunlight related to sugar?".

```
question(q).
has(q, type, how).
has(q, category, relation).
has(q, param1, sunlight).
has(q, param2, sugar).
```

Encoding a Lowest Common Ancestor

In the following is the encoding of a lowest common ancestor of two nodes. The first rule is to find all ancestors Z of X and Y. The second and third rule state that $Z2$ is a lowest common ancestor (of X and Y) if another common ancestor $Z2$, which is a descendant of $Z1$, does not exist.

```
l1: common_ancesstor(Z,X,Y) :- cpath_connects2(Z, X),
   cpath_connects2(Z, Y), X != Y.
l2: not_lcs(Z1, X, Y) :- common_ancesstor(Z1,X,Y), common_ancesstor(
   Z2,X,Y), Z1 != Z2, cpath_connects(Z1, Z2).
l3: lcs(Z, X, Y) :- common_ancesstor(Z, X, Y), not not_lcs(Z, X, Y).
```

Definition 53 *The ASP program Π_{lca} is the answer set program consisting of rules l1 to l3.*

Encoding the MIN_KDG Structure and Adding Nodes of a MIN_KDG to the Answer

In the following is the encoding of the MIN_KDG structure and how nodes of the MIN_KDG are added to the answer.

```
z4:   min_kdg(Q, Z, LCS, X, Y) :-
      question(Q),
      has(Q, type, how),
      has(Q, category, relation),
      has(Q, param1, XClass),
      has(Q, param2, YClass),
```

```
      has(X, instance_of, XClass),s
      has(Y, instance_of, YClass),
      lcs(LCS, X, Y),
      context(Z).

p0: step(0..numstep).
p5: selected_path3(Z, LCS, Z, 0) :- min_kdg(Q, Z, LCS, X, Y).
p6: selected_path3(LCS, X, LCS, 0) :- min_kdg(Q, Z, LCS, X, Y).
p7: selected_path3(LCS, Y, LCS, 0) :- min_kdg(Q, Z, LCS, X, Y).

p8: 1 { selected_path3(A, Z, C, T+1) : cpath_connects2(C, Z) :
   dcpath_connects(B, C) } 1 :- selected_path3(A, Z, B, T), B != Z,
   step(T).

p9: completed_path3(min_kdg(Q, Z, LCS, X, Y)) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(Z, LCS, LCS, T1), selected_path3(LCS, X, X,
    T2), selected_path3(LCS, Y, Y, T3),  step(T1), step(T2), step(T3
   ).
p10: :- min_kdg(Q, Z, LCS, X, Y), not completed_path3(min_kdg(Q, Z,
   LCS, X, Y)).

% Three cpaths:
a4: ans(Q, min_kdg(Q, Z, LCS, X, Y), node, N) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(Z, LCS, N, T), step(T).
a5: ans(Q, min_kdg(Q, Z, LCS, X, Y), node, N) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(LCS, X, N, T), step(T).
a6: ans(Q, min_kdg(Q, Z, LCS, X, Y), node, N) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(LCS, Y, N, T), step(T).

% Every node on the opath between two nodes on the 3 paths above
a7: ans(Q, MinKDG, node, N) :- ans(Q, MinKDG, node, X), ans(Q,
   MinKDG, node, Y), oconnects(X, N), oconnects(N, Y).
```

```
% Adding all the edges in the KDG between two nodes of structure G
    to the answer.
a3: ans(Q,G,edge,has(A,Predicate,B)) :- has(A,Predicate,B), ans(Q,G,
    node,A), ans(Q,G,node,B).
```

Definition 54 *The ASP program* $\Pi^3_{answer}(numstep)$ *is the answer set program consisting of the rules:*

- *z4 for defining MIN_KDG*
- *p5 to p10 and p0 for selecting a cpath*
- *a3 to a7 for selecting nodes and edges for the answer set*

where numstep is the constant in rule p0.

Corollary 8 *Let x and y be two nodes in the KDG*(z)*. l is a lowest common ancestor of x and y in the KDG*(x) *iff*

$$\Pi_G(z) \cup \Pi_{path} \cup \Pi_{lca} \models lca(z,x,y).$$

Definition 55 *The ASP program* $\Pi^3(q,z,numstep)$ *is the answer set program:*

$$\begin{aligned}\Pi^3(q,z,numstep) =& \Pi_G(z) \cup \Pi_{path} \cup \Pi_Q(q) \\ & \cup \Pi_{lca} \cup \Pi^3_{answer}(numstep)\end{aligned}$$

Proposition 12 *Let q*3 *be the question "How does xc related to yc?" and x and y respectively be instances of xc and yc in the KDG*(z)*. For each MIN_KDG*$^z_{x,y}$ *K, there exists an answer set of* $\Pi^3(q3,z,numstep)$ *containing:*

$$ans(q3, min\_kdg(q3,z,x), node, N)$$

and

$$ans(q3, min\_kdg(q3, z, x), edge, has(I, Predicate, J))$$

where numstep is equal to or greater than the length of longest cpath among the three cpaths from z to LCA(x,y), from LCA(x,y) to x, and from LCA(x,y) to x. N is any node of K, and (I,Predicate,J) is any edge of K.

Proposition 13 *Let q3 be the question "How does xc related to yc?" and x and y respectively be instances of xc and yc in the KDG(z). If A is an answer set of* $\Pi^3(q3, z, numstep)$*, let*

$V = \{n : ans(q3, min\_kdg(q3, z, x), node, n) \in A\}$*, and*

$E = \{(i, Predicate, j) : ans(q3, min\_kdg(q3, z, x), edge, has(i, Predicate, j)) \in A\}$*. There exists an MIN_KDG*$^z_{x,y}$ *K so that V is K's set of nodes, and E is K's set of edges.*

5.1.5 Answering Question "How Does X Participate in Y?"

Illustrated below are encodings of the question "How does sunlight participate in photosynthesis?".

```
question(q).
has(q, type, how).
has(q, category, participation).
has(q, param1, sunlight).
has(q, param2, photosynthesis).
```

Encoding the Ppath

```
pp1: ppath_connects(X, Y) :- dpconnects(Y, X), entity(X), event(Y).
pp2: ppath_connects(X, Z) :- ppath_connects(X, Y), event(Y), has(Z,
   subevent, Y).

z5: ppath(Q, Z, X, Y) :-
```

```
    question(Q),
    has(Q, type, how),
    has(Q, category, participation),
    has(Q, param1, XClass),
    has(Q, param2, YClass),
    has(X, instance_of, XClass),
    has(Y, instance_of, YClass),
    entity(X), event(Y),
    ppath_connects(X, Y),
    context(Z).
```

Adding Nodes in the PPATH to the Answer

```
p0: step(0..numstep).
p11: selected_path4(X, Y, X, 0) :- ppath(Q, Z, X, Y).
p12: 1 { selected_path4(A, Z, B, 1) : dpconnects(B, A) } 1 :-
   selected_path4(A, Z, A, 0).
p13: 1 { selected_path4(A, Z, C, T+1) : has(C, subevent, B) } 1 :-
   selected_path4(A, Z, B, T), T >= 1, step(T), B != Z.
p14: completed_path4(ppath(Q, Z, X, Y)) :- ppath(Q, Z, X, Y),
   selected_path4(X, Y, Y, T), step(T).
p15: :- ppath(Q, Z, X, Y), not completed_path4(ppath(Q, Z, X, Y)).

a8: ans(Q,ppath(Q, Z, X, Y),node,N) :- ppath(Q, Z, X, Y),
   selected_path4(X,Y,N,T), step(T).
% Adding all the edges in the KDG between two nodes of structure G
   to the answer.
a3: ans(Q,G,edge,has(A,Predicate,B)) :- has(A,Predicate,B), ans(Q,G,
   node,A), ans(Q,G,node,B).
```

Definition 56 *The ASP program* $\Pi^4_{answer}(numstep)$ *is the answer set program consisting of the rules:*

- *pp1, pp2 for defining ppath_connects*
- *z5 for defining PPATH*
- *p11 to p15 and p0 for selecting a ppath*
- *a8 and a3 for selecting nodes and edges for the answer set*

where numstep is the constant in rule p0.

Definition 57 *The ASP program* $\Pi^4(q,z,numstep)$ *is the answer set program:*

$$\Pi^4(q,z,numstep) = \Pi_G(z) \cup \Pi_{path} \cup \Pi_Q(q) \cup \Pi^4_{answer}(numstep)$$

Proposition 14 *Let* $q4$ *be the question "How does xc participate in yc?" and x and y respectively be instances of xc and yc in the* $KDG(z)$ *so that there is a participant path from x to y (Definition 11). For each* $PPATH^z_{x,y}$ *K, there exists an answer set of* $\Pi^4(q4,z,numstep)$ *containing:*

$$ans(q4, ppath(q4,z,x), node, N)$$

and

$$ans(q4, ppath(q4,z,x), edge, has(I, Predicate, J))$$

where numstep is equal to or greater than the length of K. N is any node of K, and $(I, Predicate, J)$ *is any edge of K.*

Proposition 15 *Let* $q4$ *be the question "How does xc participate in yc?" and x and y respectively be instances of xc and yc in the* $KDG(z)$ *so that there is a participant path from x to y (Definition 11). If A is an answer set of* $\Pi^4(q4,z,numstep)$*, let* $V = \{n : ans(q4, ppath(q4,z,x), node, n) \in A\}$ *and*
$E = \{(i, Predicate, j) : ans(q4, ppath(q4,z,x), edge, has(i, Predicate, j)) \in A\}$*. There exists a* $PPATH^z_{x,y}$ *K so that V is K's set of nodes, and E is K's set of edges.*

5.1.6 Answering Question "Why is X important to Y?"

Illustrated below are the encodings of the question "Why is sunlight important to photosynthesis?".

```
question(q).
has(q, type, why).
has(q, category, important).
has(q, param1, sunlight).
has(q, param2, photosynthesis).
```

Encoding the "important path"

```
% All the cpath edges except "result" - Reversed direction
i1: dipath_connects(Y, X) :- dcpath_connects(X, Y), not has(X,
   result, Y).
% "result" - Same direction
i2: dipath_connects(X, Y) :- has(X, result, Y).
% Ordering edges - Same direction
i3: dipath_connects(X, Y) :- doconnects(X, Y).

% Transitive closure of ipath_connects
i4: ipath_connects(X, Y) :- dipath_connects(X, Y).
i5: ipath_connects(X, Z) :- ipath_connects(X, Y), dipath_connects(Y,
    Z).
```

Define the Structures in the Answer

The answer of this question contains two structures: *MIN_KDG* and *IPATH*.

```
z6: min_kdg(Q, Z, LCS, X, Y) :-
      question(Q),
      has(Q, type, why),
      has(Q, category, important),
```

```
      has(Q, param1, XClass),
      has(Q, param2, YClass),
      has(X, instance_of, XClass),
      has(Y, instance_of, YClass),
      lcs(LCS, X, Y),
      context(Z).

z7: ipath(Q, Z, X, Y) :-
      question(Q),
      has(Q, type, why),
      has(Q, category, important),
      has(Q, param1, XClass),
      has(Q, param2, YClass),
      has(X, instance_of, XClass),
      has(Y, instance_of, YClass),
      ipath_connects(X, Y),
      context(Z).
```

Adding Nodes in the IPATH to the Answer

```
p0: step(0..numstep).
% Selecting nodes in the important path
p16: selected_path5(X, Y, X, 0) :- ipath(Q, Z, X, Y).
p17: 1 { selected_path5(A, Z, C, T+1) : dipath_connects(B,C) } 1 :-
   selected_path5(A, Z, B, T), step(T), B != Z.

p18: completed_path5(ipath(Q, Z, X, Y)) :- ipath(Q, Z, X, Y),
   selected_path5(X, Y, Y, T), step(T).
p19: :- ipath(Q, Z, X, Y), not completed_path5(ipath(Q, Z, X, Y)).

a9: ans(Q,ipath(Q, Z, X, Y),node,N) :- ipath(Q, Z, X, Y),
   selected_path5(X,Y,N,T), step(T).
```

```
% Three cpaths:
a4: ans(Q, min_kdg(Q, Z, LCS, X, Y), node, N) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(Z, LCS, N, T), step(T).
a5: ans(Q, min_kdg(Q, Z, LCS, X, Y), node, N) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(LCS, X, N, T), step(T).
a6: ans(Q, min_kdg(Q, Z, LCS, X, Y), node, N) :- min_kdg(Q, Z, LCS,
   X, Y), selected_path3(LCS, Y, N, T), step(T).

% Every node on the opath between two nodes on the 3 paths above
a7: ans(Q, MinKDG, node, N) :- ans(Q, MinKDG, node, X), ans(Q,
   MinKDG, node, Y), oconnects(X, N), oconnects(N, Y).

% Adding all the edges in the KDG between two nodes of structure G
   to the answer.
a3: ans(Q,G,edge,has(A,Predicate,B)) :- has(A,Predicate,B), ans(Q,G,
   node,A), ans(Q,G,node,B).
```

Definition 58 *The ASP program* $\Pi^5_{answer}(numstep)$ *is the answer set program consisting of the rules:*

- *i1 to i5 for defining ipath_connects*
- *z6, z7 for defining MIN_KDG and IPATH*
- *p16 to p19 and p0 for selecting an ipath*
- *a9, a3-a7 for selecting nodes and edges for the answer set*

where numstep is the constant in rule p0.

Definition 59 *The ASP program* $\Pi^5(q,z,numstep)$ *is the answer set program:*

$$\Pi^5(q,z,numstep) = \Pi_G(z) \cup \Pi_{path} \cup \Pi_Q(q) \cup \Pi^5_{answer}(numstep)$$

Proposition 16 *Let q5 be the question "Why is xc important to yc?" and x and y respectively be instances of xc and yc in the KDG(z) so that there is a* $IPATH^z_{x,y}$. *For each K =* $IPATH^z_{x,y} \cup MIN\_KDG^z_{x,y}$ *, there exists an answer set of* $\Pi^5(q5, z, numstep)$ *containing all of the following:*

1.

$$ans(q5, ipath(q5, x, y), node, N)$$

or

$$ans(q5, min\_kdg(q5, z, lcs, x, y), node, N)$$

2.

$$ans(q5, ipath(q5, x, y), edge, has(I, Predicate, J))$$

or

$$ans(q5, min\_kdg(q5, z, lcs, x, y), edge, has(I, Predicate, J))$$

where numstep is equal to or greater than the maximum length of the $IPATH^z_{x,y}$, *and the three cpaths composing* $MIN\_KDG^z_{x,y}$ *(cpath from z to* $LCA(x,y)$ *and from* $LCA(x,y)$ *to x, y); N is any node of K.* $(I, Predicate, J)$ *is any edge of K.*

Proposition 17 *Let q5 be the question "Why is xc important to yc?" and x and y respectively be instances of xc and yc in the KDG(z) so that there is a* $IPATH^z_{x,y}$. *If A is an answer set of* $\Pi^5(q5, z, numstep)$, *let*

$$V_1 = \{n : ans(q5, ipath(q5, x, y), node, n) \in A\}$$

$$V_2 = \{n : ans(q5, min\_kdg(q5, z, , x, y), node, n) \in A\}$$

$$E_1 = \{(i, Predicate, j) : ans(q5, ipath(q5, x, y), edge, has(i, Predicate, j)) \in A\}$$

$$E_2 = \{(i, Predicate, j) : ans(q5, min\_kdg(q5, z, , x, y), edge, has(i, Predicate, j)) \in A\}$$

There exists an K = $IPATH^{z}_{x,y} \cup MIN\_KDG^{z}_{x,y}$ so that $V_1 \cup V_2$ is K's set of nodes, and $E_1 \cup E_2$ is K's set of edges.

Chapter 6

TRANSLATING NATURAL LANGUAGE TO FORMAL LANGUAGE

In previous chapters, we discussed various aspects of answering deep questions with respect to a knowledge base. In this chapter, we will discuss extending our research to use natural language text. This includes translating deep questions in natural language to some formal representation, and a long-term target of translating natural language text to KDG specifications. We will focus on NL2KR, a system we used to address the first target, although it was developed for a broader purpose: translating natural language to formal language. We will also give a glimpse of Knowledge Parser, our first attempt to translate natural language to KDG specifications, which is not included in this dissertation.

6.1 Introduction and Motivation

Most tasks involving Natural Language Understanding (NLU) need translation of natural language (NL) to a formal language that can be further processed. Recently, algorithms have been devised to learn such semantic parsers for many applications, including command interpretations (Ge and Mooney, 2009; Chen and Mooney, 2011; Artzi and Zettlemoyer, 2013b), human robot interactions (Dzifcak *et al.*, 2009; Matuszek *et al.*, 2013; Krishnamurthy and Kollar, 2013), puzzle solving (Baral and Dzifcak, 2012), question answering (QA) (Kwiatkowski *et al.*, 2011; Liang *et al.*, 2013), query construction (Krishnamurthy and Mitchell, 2012), and program generation (Kushman and Barzilay, 2013). While outputs of semantic parsers can serve that role for specific applications, it would be preferable to build a platform that can learn such a translation system given an application. In this research, we present such a platform, called NL2KR. It is a user-friendly platform that takes sentences and their translations (language can vary depending on the application) and

some initial dictionary (for bootstrapping) to construct a translation system for the desired target language (from natural language input).

Our approach to translating natural language text to formal representation is inspired by Montague's work (Montague, 1974), where the meanings of words and phrases are expressed as λ-calculus expressions. The meaning of a sentence is built from semantics of constituent words through appropriate λ-calculus applications. A key impediment in using this approach has been the difficulty of coming up with the λ-calculus representations of words.

Although Montague's approach has been widely used (Blackburn and Bos, 2005; Zettlemoyer and Collins, 2005; Costantini and Paolucci, 2010; Baral *et al.*, 2011; Kwiatkowski *et al.*, 2010, 2013) to translate natural language to formal languages, to our knowledge, there has not been any other user-friendly platform that allows users to create their own translation system. The closest system to ours is the Semantic Parsing Framework (UW SPF) (Artzi and Zettlemoyer, 2013a).

UW SPF's approach to overcoming the impediment is to try all arbitrary splits of the input sentence and all arbitrary splits of its meaning (using an algorithm called high-order unification), then consider all combinations of the two splitting results. UW SPF does not use the CCG parser but induces the CCG parse trees to fit the current splitting. Hence, those "phrases" [1] and parse trees usually conflict with the grammatical/common sense understanding of sentences, making UW SPF learn a large number of tuples (word, CCG category, meaning) very specific to the arbitrary splitting. Most importantly, the high-order unification algorithm used to split the lambda expression imposes many restrictions (such as limited forms of functional application) on the target language, which severely limit its usefulness on new applications. Please see section 6.7 for more details.

Our NL2KR platform addresses the problem with two different parallel approaches. (1)

[1] Could be any subsequence of words in the sentence, not need to be a correct English phrase

Type	Sentence	Meaning
Training	John eats rice	eat(john,rice)
Testing	Mary drinks water	drink(mary,water)

Table 6.1: A Very Simple Corpus to Explain the Various Techniques We Are Using in NL2KR.

Using $Inverse - \lambda$ and Generalization (see section 6.2), we do not enforce the restrictions that high-order unification algorithms do. Moreover, based on known meanings, we can learn the meanings of new words, which should have higher quality. (2) Instead of generating CCG categories of words, we rely on a good CCG parser to make the tuples (word, CCG category, meaning) more general, thus reducing the number of tuples needed. Moreover, generalization and CCG parser allow NL2KR to translate sentences, which have previously unseen words. We will clarify these points using a very simple corpus that contains only one sentence for training and one for testing.

Consider the corpus shown in Table 6.1. The CCG parse trees of the two sentences in the corpus are shown in Figure 6.1 and 6.2. The CCG parse trees give us two things:

1. The words "John," "Mary," "rice," and "water" have the same CCG category; "eats" and "drinks" have the same category. This allow Generalization algorithm to work, finding the meanings of the others from one word, i.e. of "Mary" from "John".

2. The meanings of "rice" combine with ones of "eats" initially, and the meanings of two-word phrase combine with the ones of "John" to have the meanings of whole sentences.

Consider the meaning "eat(John,rice)" of the training sentence. Assume that we know that the meaning of "John" is simply "John". NL2KR will learn the meanings of other words through $Inverse - \lambda$ and Generalization; using Generalization, we can guess that

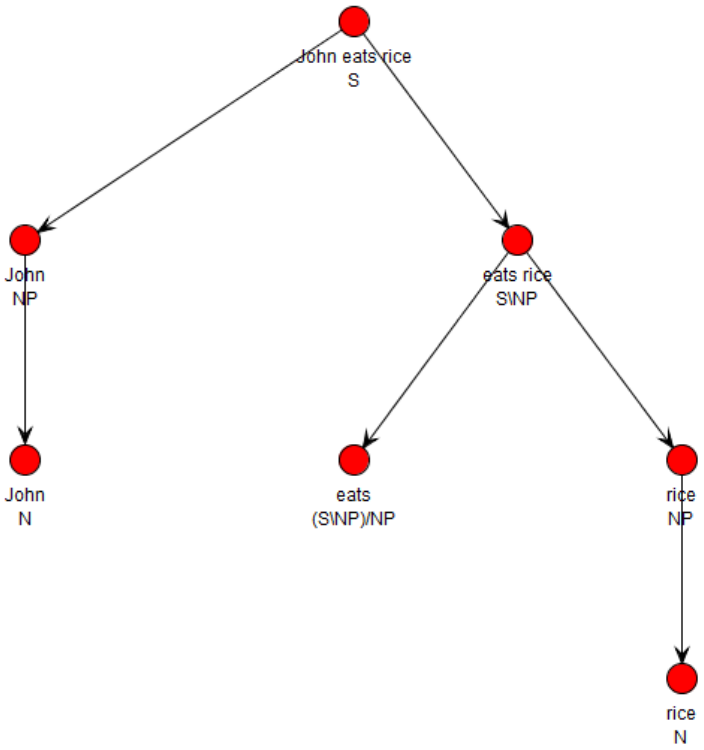

Figure 6.1: CCG parse tree of sentence "John eats rice"

"rice" has the meaning "rice;" using the $Inverse - \lambda$ algorithm, from the meaning of "John eats rice" and "John," we can get the meaning of "eats rice." Similarly, we get the meaning of "eats."

When NL2KR encounters the testing sentence, "Mary drinks water," even though the words in the sentence are unseen, CCG parse trees show that they have similar categories to known words. Additionally, Generalization can generate their meanings from known meanings, and thus can obtain the meaning of the whole sentence. For example, "water" means "water" (generalized from "rice" or "John"), "drinks" means "drink(,)" (this is not exact meaning. Its exact meaning in λ-calculus will be shown in the next chapter).

We have evaluated our platform on two datasets: GeoQuery (Zelle and Mooney, 1996) and Jobs (Tang and Mooney, 2001). GeoQuery is a database of geographical information.

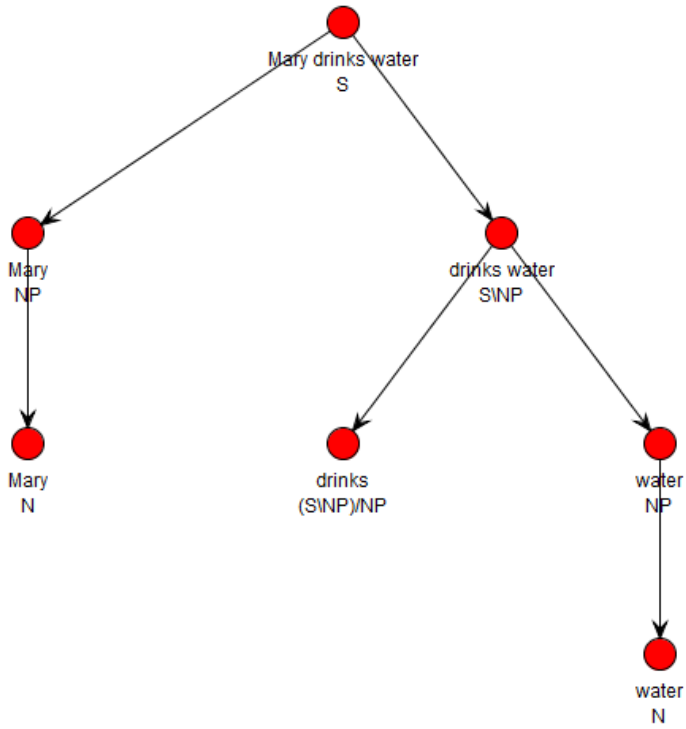

Figure 6.2: CCG parse tree of sentence "Mary drinks water"

Jobs contains sentences with job related query. Experiments demonstrate state-of-the-art performance in both cases with fairly small initial dictionaries.

The rest of the chapter is organized as follows: we first present the architecture and algorithms of NL2KR platform, then discuss the experiments. We, then, will present some take-out lessons of effective NL2KR use. After that, we will discuss about Knowledge Parser, and conclude our contributions in NL2KR.

6.2 Background

6.2.1 Lambda Calculus

Lambda calculus was proposed by Alonzo Church (Church, 1936) to investigate functions, function application, and recursion. Lambda expressions can be thought of in terms of functions and arguments. $\lambda x.bird(x)$ is an example of a lambda expression which ab-

stracts over x and signifies that the predicate bird accepts a single argument or input x. If we apply this function on *tweety* as input, denoted by $\lambda x.bird(x)@tweety$, we obtain $bird(tweety)$. This way of applying one lambda expression to another is particularly useful to us in constructing meaning from words or phrases in a sentence (explained in Sec. 6.2.2).

Let us assume V is an infinite but fixed set of identifiers. Formally, a λ-expression is one of the following: (1) a variable v in V, (2) an abstraction $(\lambda v.e)$ where v is a variable and e is a λ-expression, (3) an application $e1@e2$ where $e1$ and $e2$ are two λ-expressions, or (4) a constant. For example, x, $\lambda x.boxer(x)$, $fly(tweety)$, and *john* are all λ-expressions. Variables in a λ-expression can be bound or free. In the example expressions, x is free in x and bound in $\lambda x.boxer(x)$.

There are some operations defined over λ expressions e.g. α-conversion and β-reduction. α-conversion is the operation of renaming bound variables in a λ-expression to yield an equivalent expression. For examples, changing $\lambda x.plane(x)$ to $\lambda u.plane(u)$ does not change the meaning of the expression. β-reduction is the operation of applying functions to their arguments by replacing all instances of a free variable v in the function by an argument expression R, e.g. $\lambda x.fly(x)@tweety$ yields $fly(tweety)$.

6.2.2 Montague's Approach

Our work is inspired by Montague Semantics (Montague 1974) which postulates that the meaning of a sentence is built from meanings of its parts, which combine through syntactic operations. The semantic meaning of words and phrases are expressed in terms of λ-calculus expressions, and the meaning of the whole sentence is constructed from constituent words through successive λ-calculus applications. Montague's approach has been used in many works including (Blackburn and Bos, 2005; Zettlemoyer and Collins, 2005; Baral *et al.*, 2008; Dzifcak *et al.*, 2009; Costantini and Paolucci, 2010; Baral *et al.*, 2012a;

Zettlemoyer and Collins, 2007)

Consider the following two examples to illustrate the use of λ-calculus in translating English sentences to first order logic.

Example 1:

"John eats rice" can be translated to "eat(john, rice)" using the following λ-expressions:

- "eats": $\lambda u \lambda v.eat(v,u)$
- "rice": $rice$
- "eats rice": its meaning is obtained by the λ-application (denoted by @) between "rice" and "eats". As shown by CCG categories in the sentence's CCG parse tree (more about this in the next section), "rice" will be the argument and "eats" will be the function in the λ-application.

  "eats rice" = $\lambda u \lambda v.eat(v,u)@rice = \lambda v.eat(v,rice)$.

  The leading λu of $\lambda u \lambda v.eat(v,u)$ is removed and all other occurrence(s) of u are replaced by $rice$.
- "John" : $john$
- "John eats rice": its meaning is obtained by the λ-application between "eats rice" and "John," where "John" is the argument.

  "John eats rice": $\lambda v.eat(v,rice)@john = eat(john,rice)$

  Here the leading λv of $\lambda v.eat(v,rice)$ is removed and all other occurrence(s) of v are replaced by $john$.

Example 2:

Another example is the sentence "Most birds fly," which can be translated into first order logic as $fly(X) \leftarrow bird(X), not \neg fly(X)$. Consider the following λ-expressions for each word:

- "Most": $\lambda u \lambda v.(v@X \leftarrow \lambda u@X, not\ \neg v@X)$.
- "birds": $\lambda x.bird(x)$
- "fly": $\lambda y.fly(y)$

The meaning of the entire sentence in terms of λ-calculus expressions can be obtained by combining its parts as follows:

- "Most birds": $(\lambda u \lambda v.(v@X \leftarrow \lambda u@X, not \neg v@X))@(\lambda x.bird(x))$ which simplifies to $\lambda v.(v@X \leftarrow bird(X), not \neg v@X)$.
- "Most birds fly" : $(\lambda x.x@\lambda y.fly(y))@(\lambda v.(v@X \leftarrow bird(X), not \neg v@X))$ which simplifies to $fly(X) \leftarrow bird(X), not \neg fly(X)$.

In order to use Montague's approach, we need to know the following:

1. The order of combination of sentence constituents and rules for their combination
2. Semantic meanings of words in the sentence in terms of λ-expressions

A Combinatory Categorial Grammar(CCG)(Sec. 6.2.3) is used to find the order of combination of parts of a sentence and to define the rules for their combination. Application of this approach also requires that lambda expressions of all words in the sentence need to be known, but manually formulating them is a tedious task, especially if they are complex. The Inverse Lambda and Generalization algorithms (Sec. 6.2.4 and Sec. 6.2.5) can be used to automatically obtain these semantics from others that are known.

6.2.3 Combinatory Categorial Grammar

In the previous section, we pointed out that to build the meaning of a complete sentence, the order of combination of its parts and the rules for this combination must be known. To combine two given constituents with given λ semantics by application (or β-reduction),

we need to know which of the constituents will be the function and which will be the argument for this application. Combinatory Categorial Grammar(CCG)(Steedman, 2000) can provide this kind of information. CCG has a transparent mapping between syntax and semantics unlike other grammars, and giving compositional semantics is straightforward (Clark and Curran, 2007). In building λ expressions, the CCG parser output would be able to correctly dictate which λ expression should be applied to which. CCG is characterized by CCG categories and syntactic rules.

- **CCG Categories:** Every lexical element is assigned to at least one category.
  - **Basic categories** such as S, N, or NP,
  - **Derived categories** constructed from basic categories such as $S\backslash NP$ or NP/N, and
- **Syntactic rules** describing the concatenation operations of categories and the resulting category. For example, if X has category A/B and Y has category B, then the forward application between X and Y will yield the category A (for the phrase XY)
  - Application
    - Forward Application(>) $\frac{A/B \quad B}{A}$
    - Backward Application(<) $\frac{B \quad A\backslash B}{A}$
  - Composition
    - Forward Composition(>B) $\frac{A/B \quad B/C}{A/C}$
    - Backward Composition(<B) $\frac{B\backslash C \quad A\backslash B}{A\backslash C}$
  - Type Raising
    - Forward Type Raising(>T) $\frac{A}{B/(B\backslash A)}$
    - Backward Type Raising(<T) $\frac{A}{B\backslash(B/A)}$

CCG Syntactic rule	NL2KR operation	X's role	Y's role
Forward Application	λ-application	function	argument
Backward Application	λ-application	argument	function
Forward Composition	λ-application	function	argument
Backward Composition	λ-application	argument	function
Type Raising	keep the same λ-expression	N/A	N/A

Table 6.2: Mapping Between CCG Syntatic Rules and the Operations in NL2KR

Intuitively, the word "walks" has the category $S\backslash NP$, meaning that if an NP (a noun phrase) is concatenated to the left of "walks," then we obtain a string of category S (a sentence). Indeed, when we concatenate "John," an NP, to the left of "walks," we obtain "John walks," which is a sentence.

Table 6.2 shows the mapping between CCG syntatic rules between X and Y, and the λ-applications NL2KR used to combine the meanings of X and Y.

Table 6.3 shows an example how CCG can be used to find the order of combination of constituent words or phrases in the sentence "Most birds fly." The CCG categories of all words has been indicated. The category of "birds" is N, meaning it is a noun. The category of "Most" is N/N, signifying a modifier of a noun. The category N/N also indicates that if a noun is concatenated to the right of "most," then the category of the resulting phrase is N. Similarly, the category of "fly" is $S\backslash NP$, which is common for most intransitive verbs. It also means that "fly" will form a complete sentence (S) after concatenating with a noun phrase (NP) from the left. In this way, we know that one possible way of combination is that "most" and "birds" combine first to yield a phrase "most birds" of category N, as dictated by the combination rules of CCG. The λ semantics of this phrase is obtained by an application of the λ expression "most" as a function to the λ expression "birds" as an

Table 6.3: CCG Parse : "Most birds fly"

Most	birds	fly
N/N	N	$S\backslash NP$
$\lambda u\lambda v.(v@X \leftarrow \lambda u@X, not\ \neg v@X)$	$\lambda x.bird(x)$	$\lambda x.x@\lambda y.fly(y)$
———————————————		>
N		$S\backslash NP$
NP		$S\backslash NP$
$(\lambda u\lambda v.(v@X \leftarrow \lambda u@X, not\neg v@X))@(\lambda x.bird(x))$		
———————————————		<
S		
$(\lambda x.x@\lambda y.fly(y))@(\lambda v.(v@X \leftarrow bird(X), not\neg v@X))$		
$= fly(X) \leftarrow bird(X), not\neg fly(X).$		

argument. At the next level, this phrase combines with "fly" to yield the meaning of the entire sentence as shown. A sentence can have multiple parses.

6.2.4 Inverse Lambda Algorithm

Inverse λ Algorithm is used to learn lambda expressions for words present in training sentences. It is useful when we know the λ-expression of a phrase and the λ-expression of one of its sub-parts (ie. one of its children in the CCG parse tree), but not of the other.

NL2KR's inverse λ operation is based on the work proposed in (Gonzalez, 2010). It computes a λ-expression F such that $H = F@G$ or $H = G@F$ if given H and G. These are called Inverse-L and Inverse-R algorithms respectively.

In the example of "Most birds fly,"using λ-application (and β-reduction), we can get

the λ-expression of the whole sentence when we have those of "Most birds" and "fly," along with the CCG parse tree of the sentence. However, when the λ-expressions of "Most birds" and "Most" are unknown, the λ calculus operations do not help in finding them. In this case, Inverse λ algorithm can compute the λ-expression of "Most birds" given the λ-expressions of "fly" and that of the whole sentence. Subsequently, from the λ- expression of "Most birds," it can compute the λ- expression of "Most," given the λ-expression of "birds."

6.2.5 Generalization Algorithm

Generalization (Baral *et al.*, 2011; Kwiatkowski *et al.*, 2011) is used to learn the meanings of unknown words from similar words with known meanings. It is used when Inverse-λ is not enough to learn new meanings of words. Generalization helps NL2KR learn meanings of words that are not even present in the training data set. As an example, if we want to generalize the meaning of the word "drinks" with CCG category $(S\backslash NP)/NP$), and the lexicon already contains an entry for "eats" with the same CCG category and meaning $\lambda y.\lambda x.eat(x,y)$, the algorithm will extract the template (i.e. $\lambda y.\lambda x.PLACEHOLDER(x,y)$) and apply the template to *drinks* to get the meaning $\lambda y.\lambda x.drink(x,y)$. NL2KR can recognize if we need to convert the word to lowercase and/or to its simple form in applying the template. For example, for the word "eats," NL2KR found that we need to convert "eats" to its simple form "eat;" it will do the same with "drinks" to "drink" (not "drinks") in $\lambda y.\lambda x.drink(x,y)$.

6.3 Architecture

The NL2KR system has two sub-parts, which depend on each other: (1) NL2KR-L for learning and (2) NL2KR-T for translating, as shown in Figure 6.3.

The NL2KR-L module takes an initial lexicon consisting of some words and their meanings in terms of λ-calculus expressions and a set of training sentences, along with their

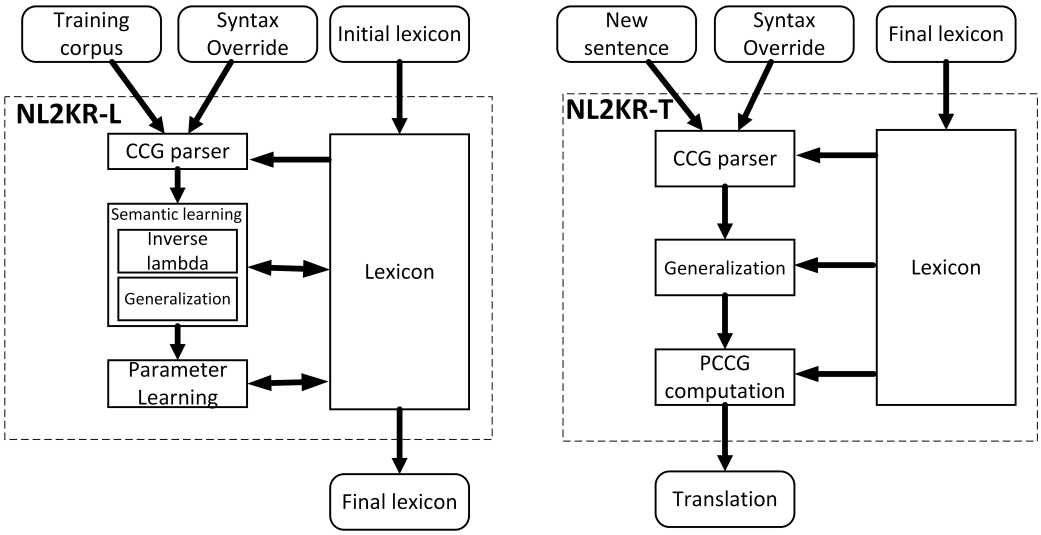

Figure 6.3: Architecture of NL2KR-L for Semantic Learning (left) and NL2KR-T for Translation (right)

target formal representations as the input. It then uses a CCG parser to construct the parse trees of the training sentences in English. Next, the learning sub-part of the system uses Inverse-λ and Generalization algorithms (Baral *et al.*, 2011) to learn meanings of words, which are either not present in the initial lexicon or have alternate meanings. A parameter learning method is then used to estimate a weight for each lexicon entry (word, its CCG syntactic category and meaning), so that the joint probability of the sentences in the training set being translated to their given formal representation is maximized. Basic translation methodology is based on Probabilistic Combinatorial Categorical Grammars (PCCG) (Zettlemoyer and Collins, 2005, 2007, 2009; Baral *et al.*, 2011). The result of NL2KR-L is a new lexicon, which contains a larger set of words, theirs meanings, and their weights.

The translation module (NL2KR-T) uses the lexicon output by the Learning module and translates sentences in test data set using the PCCG parser. Meanings of each word used in translation are either retrieved directly from the learning lexicon or generalized from it. Since words can have multiple meanings and associated λ-calculus expressions, weights

assigned to each lexical entry in the lexicon helps in deciding the more likely meaning of a word in the context of a sentence.

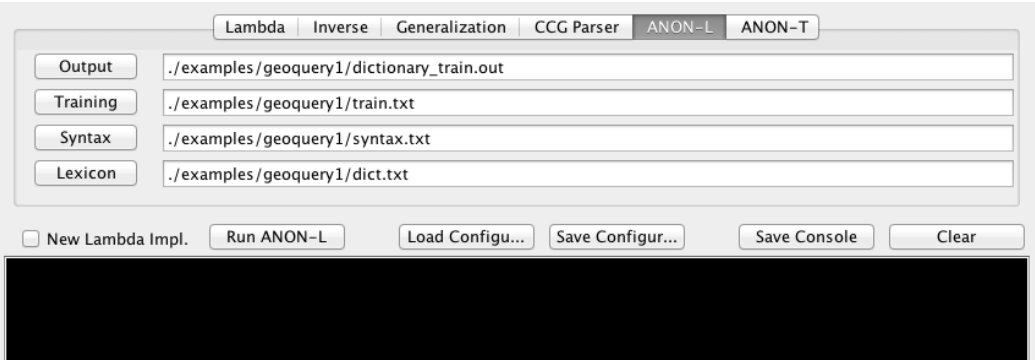

Figure 6.4: NL2KR's main GUI

6.3.1 Training Corpus

The training corpus is provided to the system as a text file with each line containing a tab-separated pair of sentences in natural language and its corresponding translations in the desired target formal language.

$$< NLSentence > \backslash t < FLSentence >$$

Example:

```
Vincent loves Mia   loves(vincent, mia)
John walks    walk(john)
```

NL2KR supports two syntaxes of the target language (Table 6.4 and Table 6.5), which users can select by checking the check box on NL2KR's main GUI. The first syntax is similar to first-order logic, while the second one can be used to represent a broader range of languages.

In some cases, it might be hard to convert directly to some target languages. For example, a procedural language like C. In such cases, we might have to use a simpler intermedi-

Syntax	Meaning	Example
$Function\_name(x_1, x_2, ...)$	function Function_name of variables $x_1, x_2, ...$ The order of $x_1, x_2, ...$ can not be changed	$job(x)$
$\sharp x.y$	$\lambda x.y$	$\sharp x.job(x)$
$x@y$	$x@y$	$\sharp x.job(x)@y = (\lambda x.job(x))@y$
$Ax(y)$	$\forall x(y)$	$Ax(boxer(x)) = \forall x(boxer(x))$
$Ex(y)$	$\exists x(y)$	$Ex(boxer(x)) = \exists x(boxer(x))$

Table 6.4: Syntax I of the Target Language

ate language as the target language, which can be later directly converted to the originally required language.

6.3.2 Initial Lexicon

The initial lexicon or dictionary is a text file with each line containing a lexical entry. A lexical entry is a tab-separated sequence of the word, its CCG category, and its semantic representation in the form of a λ-calculus expression, also referred to as its meaning.

$$< word > \backslash t < CCGcategory > \backslash t < \lambda - expression >$$

Example:

```
boxer    N      #x. boxer(x)
walks    S\NP   #x. walks(x)
```

Syntax	Meaning	Example
$f(Function\_name, x_1, x_2, ...)$	function Function_name of variables $x_1, x_2, ...$ The order of $x_1, x_2, ...$ can not be changed	$f(job, x)$ means $job(x)$
$g(Function\_name, x_1, x_2, ...)$	function Function_name of variables $x_1, x_2, ...$ The order of $x_1, x_2, ...$ cannot be changed	$g(and, x, y, z)$ = $g(and, z, x, y)$ means $x \wedge y \wedge z$
$l(x, y)$	$\lambda x.y$	$l(x, f(job, x))$
$a(x, y)$	$x@y$	$a(l(x, f(job, x)), y)$ = $(\lambda x.job(x))@y$
$f(forall, x, y)$	$\forall x(y)$	$f(forall, x, f(boxer, x))$ = $\forall x(boxer(x))$
$f(thereexist, x, y)$	$\exists x(y)$	$f(forall, x, f(boxer, x))$ = $\exists x(boxer(x))$

Table 6.5: Syntax II of the Target Language

6.3.3 Syntax Override

The Syntax Override is an optional text file, each line of which contains a tab-separated pair of a word and its CCG category. The role of the syntax override file is to override the category for that word in the CCG parse.

$$< word > \backslash t < CCGcategory >$$

Example:

```
walk  (S\NP)/NP
```

In the above example, "walk" is always forced to be a transitive verb, which takes noun phrase inputs to its left and right.

6.3.4 Learning Interface

A screenshot of the learning interface has been presented in Figure 6.5. The system takes the file paths of the training corpus, initial lexicon, and any syntax overrides as input. Then, it produces a newly learned lexicon as output at the path specified in the text box, labeled "Output". If the user does not wish to specify any syntax overrides, a blank text file may be supplied. The "Run NL2KR-L" button starts the learning process using the algorithm explained in Sec. 6.4.2. Experiment configurations may be saved; previously saved configurations may be retrieved using the "Save Configuration" and "Load Configuration" buttons respectively. The "Save console" button saves the output log.

6.3.5 Translation Interface

The translation interface takes the file path of the test data in the "Data" field and the lexicon output from the Learning phase in the "Lexicon." It also takes the location of the syntax override file provided by the user as input. Pressing "Run NL2KR-T" starts the translation process. A screenshot of the system is shown in 6.6.

6.3.6 CCG Parser

Figure 6.7 shows the output of NL2KR's CCG parser on the input sentence "Every boxer walks." Sentences can be specified individually as shown or as a batch file. The parser also takes the location of user-specified syntax overrides as input. The parse tree can be zoomed in/out, and its nodes can be moved around, making it easier to work with complex parse trees.

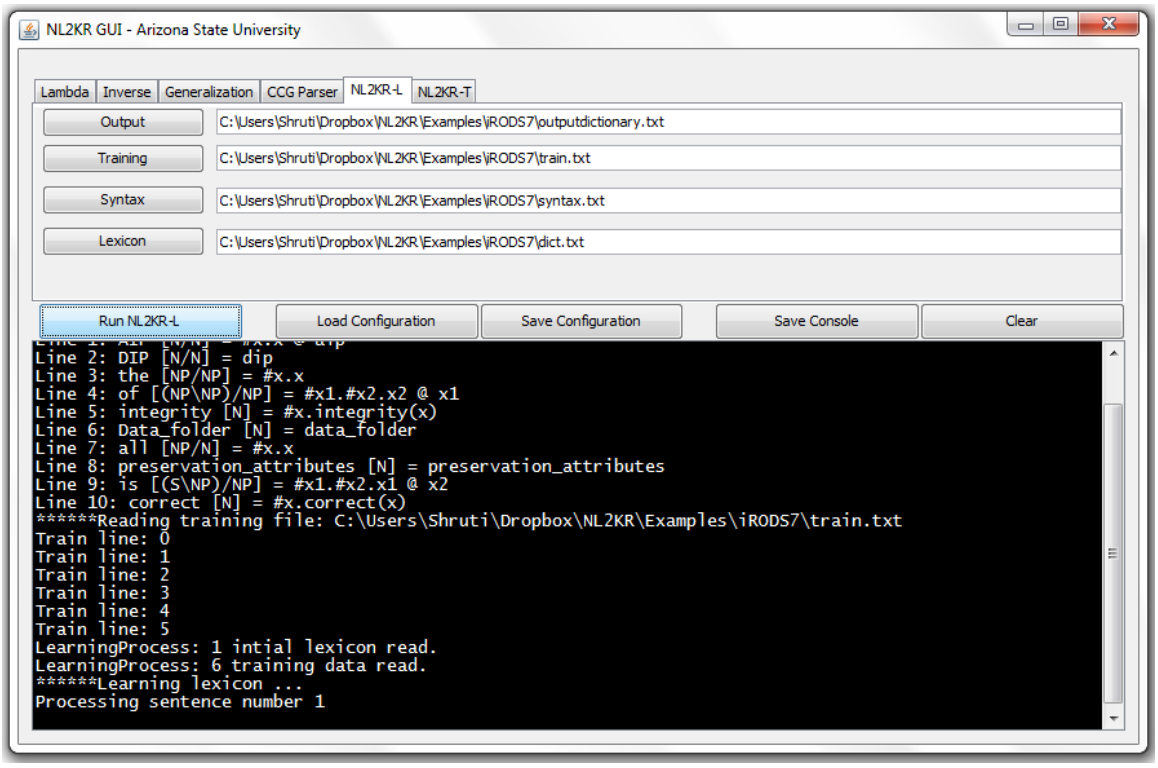

Figure 6.5: Screenshot of the Learning component of NL2KR. The system takes a training corpus, initial lexicon, and any syntax overrides as input. Then, it produces a newly learned lexicon as output at the path specified.

6.3.7 Availability and Dependency

The latest version of NL2KR system can be freely downloaded from http://bioai.lab.asu.edu/nl2kr. The User Manual and examples of using the system is also available at the downloading site.

6.4 Algorithms

6.4.1 CCG Parsing

The parse of a sentence contains information about the syntactic structure of a sentence, according to a particular grammar. A CCG parser gives us the CCG category of each word in the sentence and a parse tree which shows how they combine according to the rules of

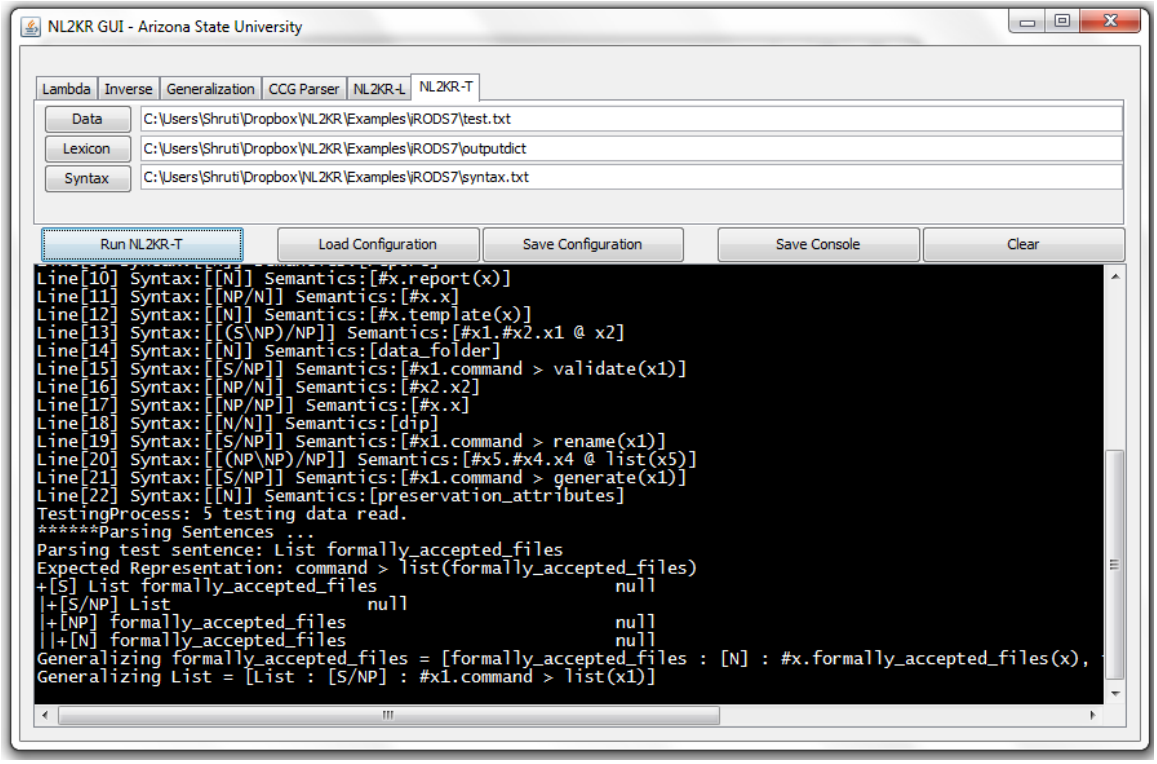

Figure 6.6: Screenshot of the Translation component of NL2KR. The system takes a test corpus, lexicon output from the Learning phase, and any syntax overrides as input. Then, it provides a log of the translation process on the console.

CCG. Figure 6.11 shows an example output of a CCG parser. In section 6.2.3, we saw how we can use CCG to determine the order of combination of parts in a sentence and ascertain which part should be the function and which the argument for their combination by lambda application. CCG parsing module is used in both the learning and translation phases of NL2KR. It can be used as a stand-alone CCG parser; the output graphs look like the one in Figure Figure 6.11. It can be zoomed in/out, and its nodes can be moved around, making it easier to work with complex sentences.

The C&C parser (Curran *et al.*, 2007) consists of a supertagger that assigns each word of the input sentence to a CCG category. The model for this tagger has been obtained by

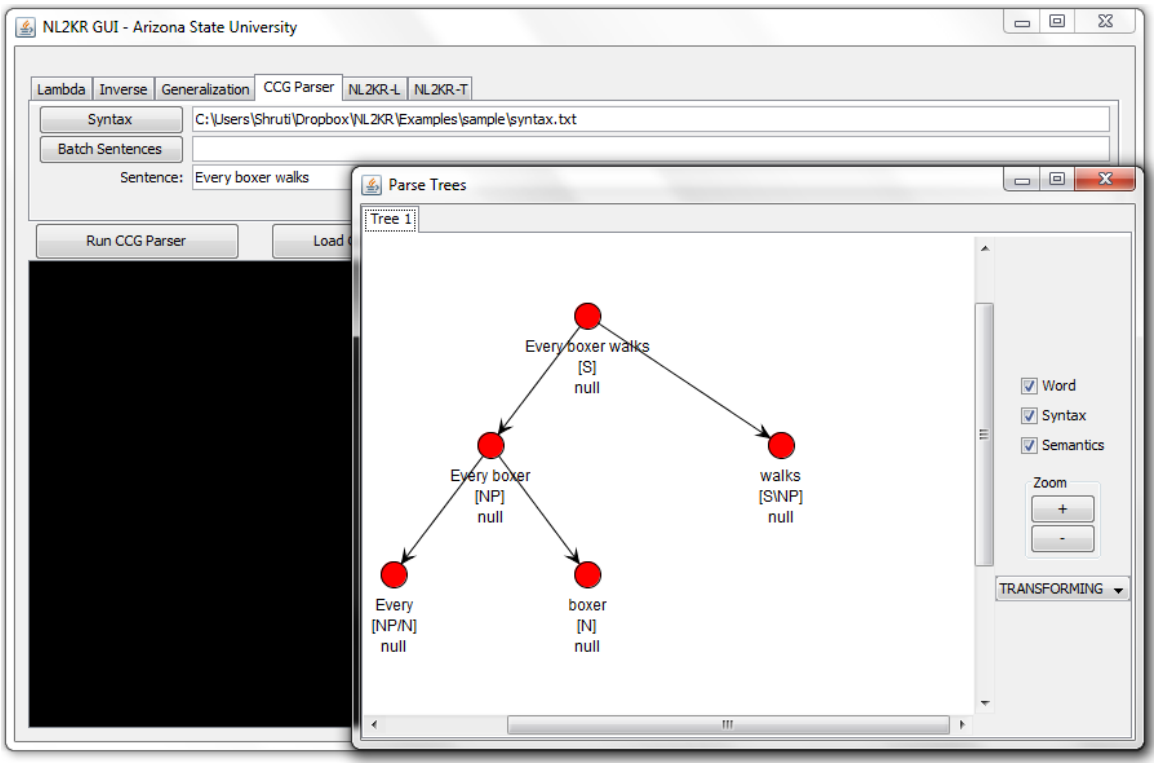

Figure 6.7: CCG Parser Screenshot.

training on the CCGbank (Hockenmaier and Steedman, 2007) corpus [2] . The C&C parser gives a single best parse tree for each input sentence. This is sometimes a limitation because one sentence can be parsed in many ways. An example of such a sentence is "He saw the astronomer with a telescope." Since we need our system to learn as many new words as possible, we want to consider multiple good parses instead of just one. The ASPCGTK parser (Lierler and Schüller, 2012) tries to overcome this limitation by giving all possible semantically distinct parse trees of a sentence. It initially uses C&C's POS tagger on the input sentence and then uses the C&C supertagger to get all possible CCG categories of each word in the sentence. Considering each CCG category as fluent and CCG operations (i.e. forward application, backward application as explained in section 6.2.3) as actions,

[2]The CCGbank corpus contains CCG parses of the sentences in the Penn Treebank

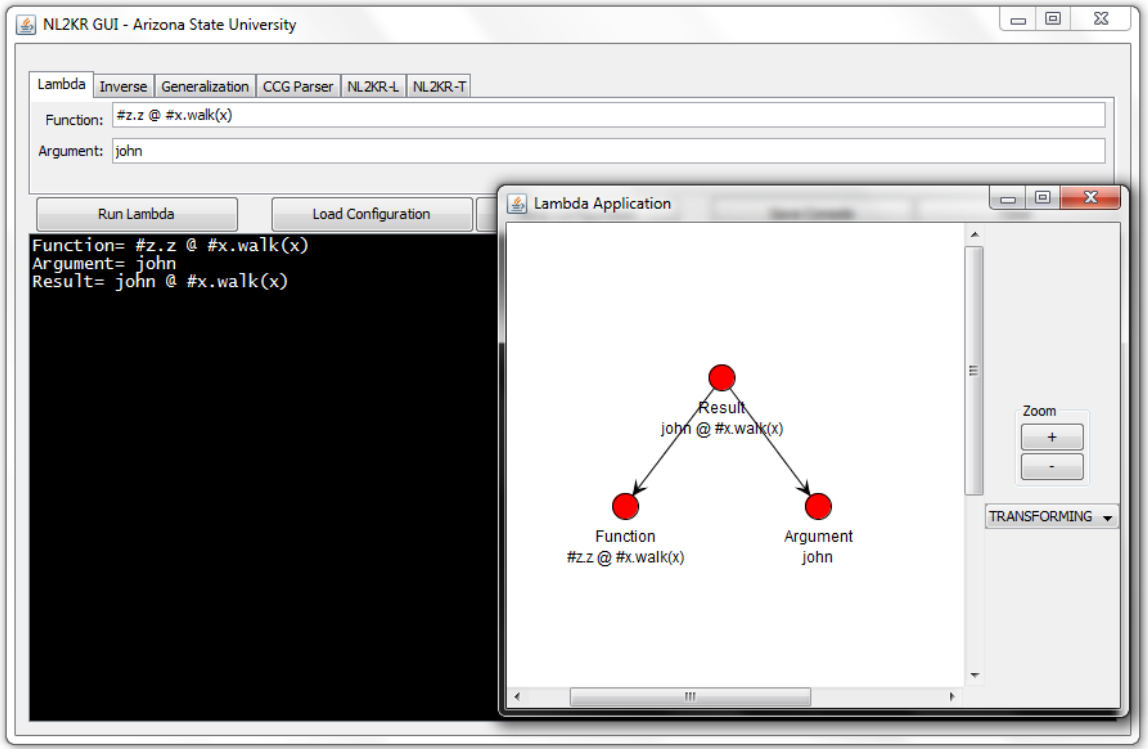

Figure 6.8: Lambda Application Screenshot.

it solves the CCG parsing problem with a simple planning solver implemented in ASP. All successful plans correspond to valid parse trees. This way, it does not generate a single best parse tree, but rather many good ones. This however falls onto the other end of the spectrum. The number of parse trees generated can range from dozens to hundreds with no preferential ordering among them. If we choose the top-k trees, and there are m trees with same score (m>k), we cannot guarantee that the same trees will be returned consistently across multiple runs. To rank trees more effectively, we need to have a more accurate scoring system.

To handle these issues, we have developed two in-house parsers possessing outputs as k weighted parse trees, where the parameter k is provided by the user. These parsers also allows overriding of CCG category for particular words, thus putting more weight on those derivations which are using overridden syntax. A particular word can have multiple

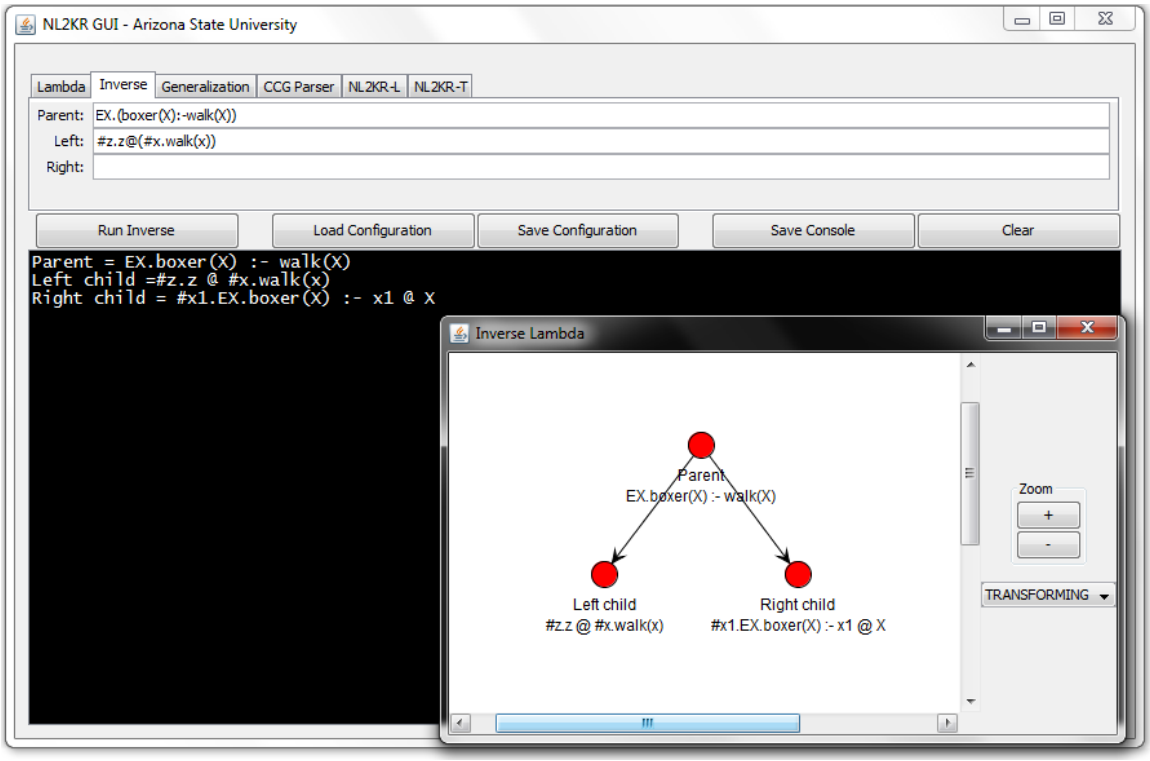

Figure 6.9: Lambda Inverse Screenshot.

overrides; in that case, all of them receive equal weighting.

The two parsers use Stanford POS tagger (Toutanova *et al.*, 2003) to assign parts-of-speech to words in the sentence. We empirically found it to be more accurate in our experiments than C&C's POS tagger. Next, instead of using C&C's supertagger direcly, they use their own supertagger to assign CCG categories to each word, which is also based on the C&C's rich model as input. Using the model directly instead of through the supertagger allows us to change the model as needed, according to domain requirements, like if the input corpus contains a large number of statements which are commands, for example. These are usually imperative statements, which are not so common in the usual corpora on which the model has been derived from. Allowing reconfiguration of the model allows us to allow proper parsing of imperative sentences as well by adding the correct category of imperative verbs in the model.

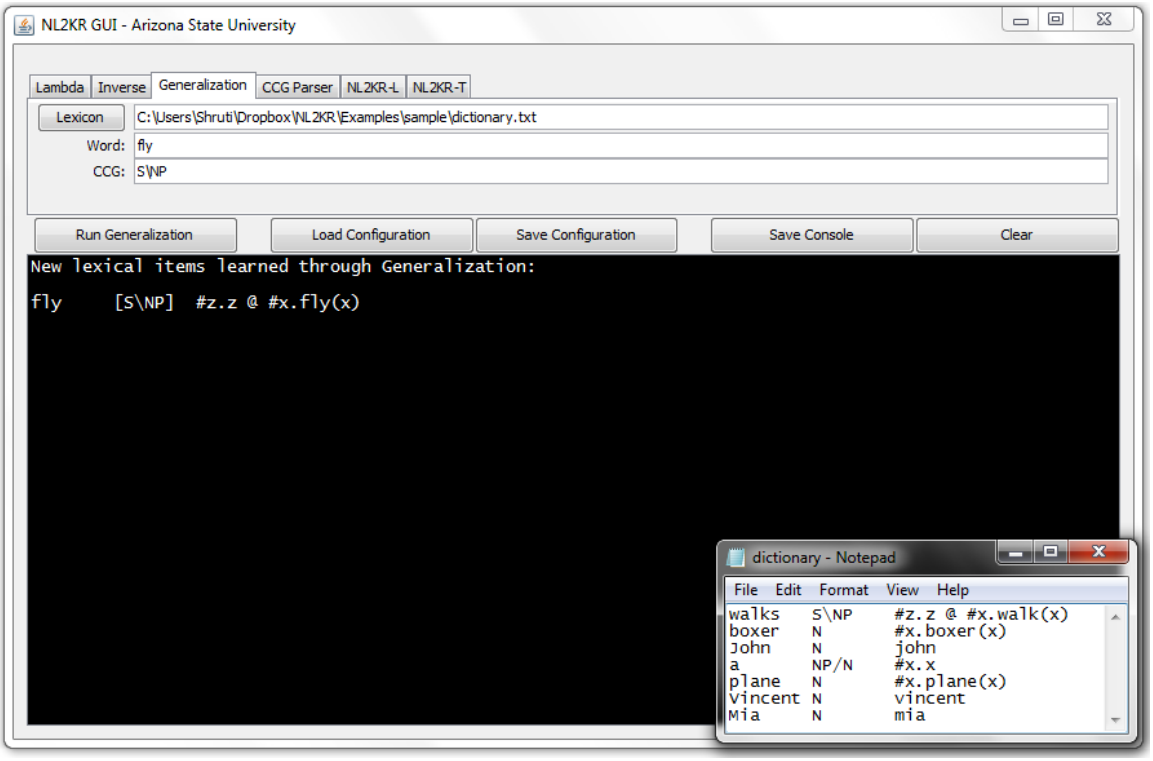

Figure 6.10: Generalization Screenshot.

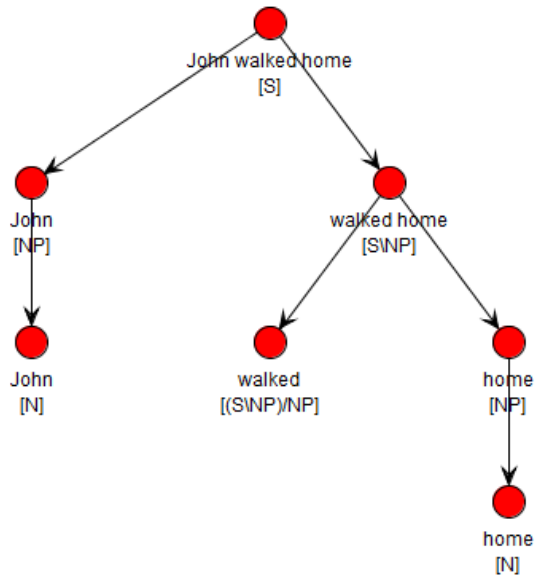

Figure 6.11: CCG parse tree of ”John walked home”.

The first parser was developed using beam search with the Cocke-Younger-Kasami(CYK) algorithm. The second one uses an idea similar to that of ASPccgTk; it also formulates the CCG parsing problem as a planning problem but represents it in Planning Domain Definition Language (PDDL) (McDermott *et al.*, 1998). While the second one is not faster than the first, it certain advantages: it is easier put additional soft constraints in CCG Parsing. These constraints could come from additional semantic parsers. For example, one constraint could be that a certain phrase is modifying a word "A," not the whole verb phrase. In the following paragraphs, we will show more details about the second parser.

CCG parsing problem as a planning problem but represents it in PDDL. The applications in CCG grammar such as the forward and backward application operations, described in PDDL as actions with preconditions and effects, have been shown in Figure 6.12 for illustration. The rest of the knowledge in CCG parsing are encoded as planning domain and fluents. The facts in a particular sentence are represented as factual input of the planning problem.

The probability of each CCG category of each word in the sentence is used to compute the total probability of the entire parse tree. The initial condition for the planning problem consists of the various CCG categories assigned to each word along with their probabilities. We then use Metric-FF (Hoffmann *et al.*, 2003), a planner solver that can handle numeric values, to solve the planning problem. Given the initial state specifying the categories of each word, all possible CCG combinatory actions, and the goal of achieving a complete sentence category (S) at the root of the parse tree(Figure 6.13), the planner finds all of the plans. These plans correspond to valid parses and have a probability score associated with them. As a result, we get a ranked list of parses. Using this solver gives us a huge advantage in speed and the ability to handle numerical values properly.

In addition, the system also provides the user the ability to customize individual parses unlike the C&C parser through the use of Syntax Overrides passed as an input into the

parser. This will give the category a probability of one for the word in the supertagger. It should be noted that the parse tree might still not have the specified category if it is not compatible with the categories of the rest of the words.

For illustration, the steps involved in obtaining the parse of "John walked home" is shown in Table 6.6. An overview of the planning process is shown in Figure 6.14.

One of the quality measures of a CCG parser is consistency. A CCG parser is consistent if the order of the weighted parse tree remains same over multiple parses of the same sentence, and sentences having similar structures have identical ordering of the derivations. This has a direct impact on the translation process. Since a derivation in CCG defines the structure of the sentence, a consistent CCG parser provides a better model for the rules, which govern how the words combine together in the target language. If sentences having similar structures are derived in different ways, the Generalization algorithm suffers, leaving a negative impact on NL2KR's performance. NL2KR's CCG Parser achieves consistency through the use of Stanford Dependency Parser and parsing algorithms. Stanford parser provides the Part Of Speech (POS) tags for the constituents in a sentence which are then fed to parsing algorithms to come up with the weighted parse trees. Parsing algorithms inherently ensure that the order of the derivations of a sentence does not change over multiple parses. Because sentences having identical structures should have identical POS tags for its constituents, NL2KR's CCG parser exhibits near-consistent behavior.

6.4.2 Multistage Learning Approach

Learning the meanings of words is the major component of our system. The inputs of the learning component are a set of training sentences, their target formal representations, and an initial lexicon consisting of some words and their meanings in terms of λ-calculus expressions. The output of algorithms is the final lexicon (dictionary), which is a list of 4-tuples (word, CCG category, meaning, weight).

```
(:action forward_application
:parameters (?x ?y ?z - category ?l1 ?l2 - location)

:precondition
    (and
        (present ?x ?l1) (present ?y ?l2)
        (next ?l1 ?l2)
        (not (resolved ?l1)) (not (resolved ?l2))
        (fa ?x ?y ?z)
    )
:effect
    (and
        (not (present ?x ?l1))
        (not (present ?y ?l2))
        (present ?z ?l1)
        (closing_needed ?l2)
        )
    )

(:action backward_application
:parameters (?x ?y ?z - category ?l1 ?l2 - location)

:precondition
    (and
        (present ?x ?l1) (present ?y ?l2)
        (next ?l1 ?l2)
        (not (resolved ?l1)) (not (resolved ?l2))
        (ba ?x ?y ?z)
    )
:effect
    (and
        (not (present ?x ?l1))
        (not (present ?y ?l2))
        (present ?z ?l1)
        (closing_needed ?l2)
    )
)
```

Figure 6.12: Forward and Backward application described as PDDL actions

Table 6.6: CCG Parsing Steps : "John walked home"

Step	**Example**
Label location of words	John(L0) walked(L1) home(L2)
Tag words with part of speech using Stanford Parser	John/NNP walked/VBD home/NN
Tag words with (multiple) corresponding CCG categories using C&C Parser's model trained on CCGBank	John: N/N(0.68), N(0.77)... walked: ((S\NP)/PP)/NP(0.42), S\NP)/NP(0.77) S\NP)/S(0.52)... home: N/N(0.21), N(0.80)...
Solve parsing as planning problem	*Initial state*: CCG categories of words *Actions*: Combinatory rules of CCG Eg. Forward and backward application, type raising etc. *Goal*: Get S as final category *Plan*: 1.SELECT (S\NP)/NP L1 2.SELECT N L0 3.SELECT N L2 4.TRANSFORM N NP L2 5.FORWARD_APPLICATION ((S\NP)/NP) NP S\NP L1 L2 6.CLOSE L1 L2 7.TRANSFORM N NP L0 8.BACKWARD_APPLICATION NP S\NP S L0 L1 9.CLOSE L0 L1
Get parse from plan	See Figure 6.11

```
(:goal
    (and
        (and
            (resolved l1) (resolved l2) (resolved l3))
        (or
            (present s l0)
            (present np l0)
        )
    )
)
```

Figure 6.13: Parsing goal defined in PDDL. Resolved is a fluent which indicates that a location has successfully combined with another to yield a new category at the other location. In this case, the goal will be achieved when the result of combination yields an *S* or a *NP* at position zero(l0) and the rest of the words (l1,l2 and l3) have been successfully combined in prior states.

Let us consider an example. Suppose the input sentence is *How big is Texas* and its CCG parse tree is shown in Fig.6.15. Each word in the sentence is represented by a node in the tree; its meaning is given right below as an λ-expression. Symbol $\sharp$ is used to represent λ. Meanings of phrases *How big* and *is Texas* are obtained by combining the meanings of their words: *How* and *big*, and *is* and *Texas*, according to the CCG grammar. For example, the CCG categories of *is* and *Texas* are (S\ NP)/NP and NP; they are combined (through forward application) to form category NP of *is Texas*. This suggests that in the lambda application to combine meanings $\sharp x.x$ and $stateid(texas)$ (of *is* and *Texas* respectively), $\sharp x.x$ is the function and the other is argument. The meaning of *is Texas* is thus $\sharp x.x@stateid(Texas) = stateid(Texas)$. Similarly, we have the meaning of *How big* is $(\sharp x.\sharp y.answer(x@y))@(\sharp x.size(x)) = \sharp y.answer((\sharp x.size(x))@y) = \sharp y.answer(size(y))$

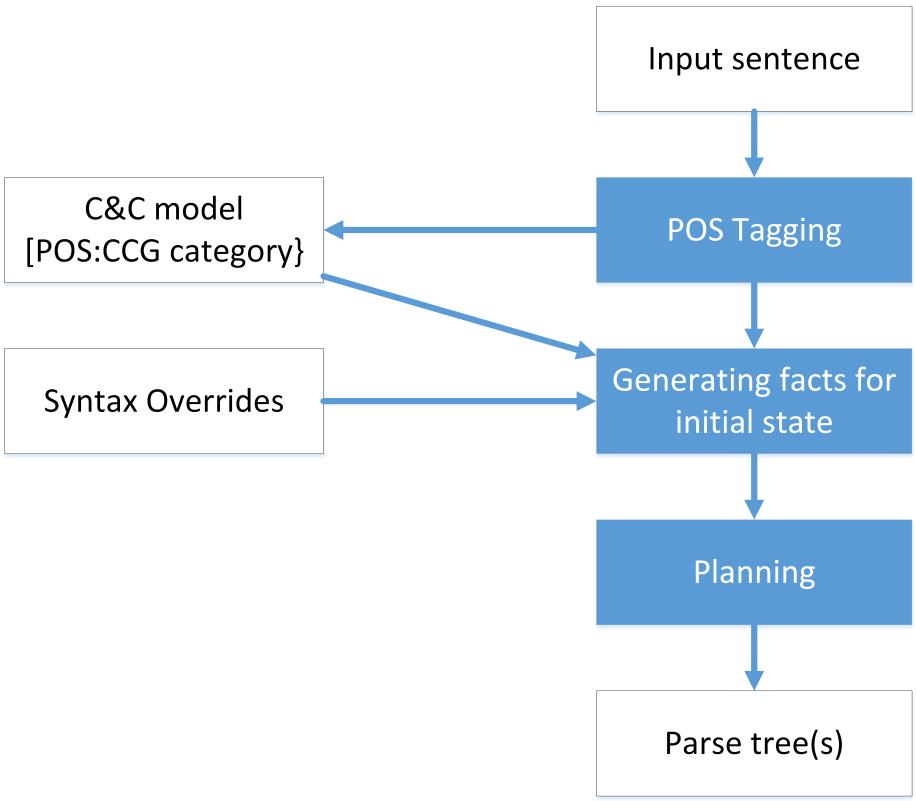

Figure 6.14: CCG Parsing Overview

When we combine *How big* and *is Texas*, the meaning $stateid(Texas)$ of *is Texas* must be used as the function, as stated by CCG. However, $stateid(Texas)$ does not have any free variables, and thus cannot be used as a function. We therefore do not have any meaning of *How big is Texas*. The meaning of the whole sentence *How big is Texas* is always given in the training set; suppose that it is $answer(size(stateid(Texas)))$. Since meanings of *How big* and *is Texas* cannot be combined to have the meaning of *How big is Texas*, at least one of them is incorrect. We can assume that one of them is correct and use the inverse lambda algorithm (also need the meaning of parent, which is *How big is texas*) to learn the other. For example, assuming the meaning $\sharp y.answer(size(y))$ of *How big* is correct; the meaning of *How big is Texas* is known as $answer(size(stateid(Texas)))$; using inverse lambda algorithm, we can calculate the new meaning of *is Texas*, which

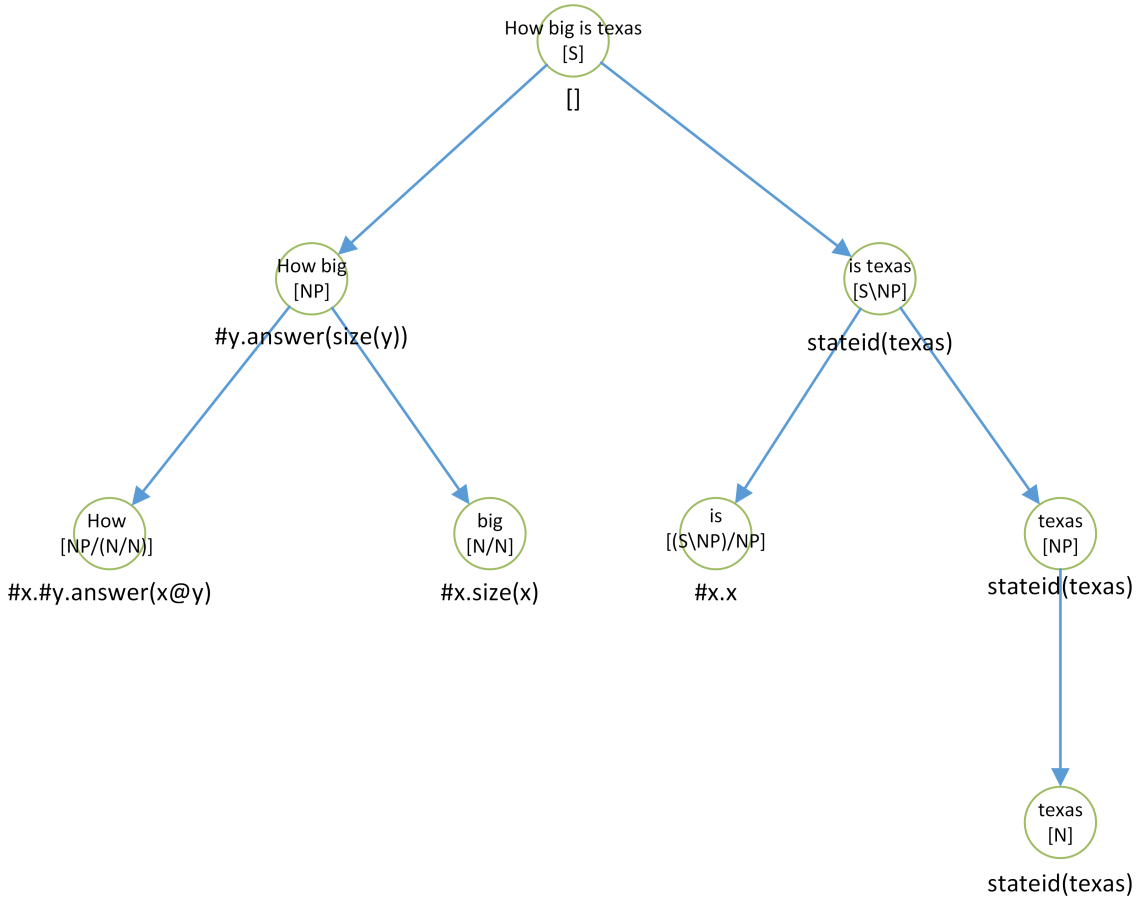

Figure 6.15: CCG Parse of "How big is texas".

is $\sharp x.x@stateid(Texas)$. Note that we can verify the correctness of this calculation by applying $\sharp y.answer(size(y))$ to $\sharp x.x@stateid(Texas)$ to get back the correct meaning of the whole sentence $answer(size(stateid(Texas)))$.

Using the new meaning of *is Texas*, we can go further down to calculate new meanings of *is* and *Texas*, and do the same thing for the branch of *How big*.

Through the example, we can come up with the simple learning idea: investigate all possible meanings of the sentence, then extensively use inverse algorithm to learn new meanings. However, if each of n words in the sentence have k meanings, we would have to check k^n combinations; each will form a parse tree similar to the one in Fig.6.15. For each

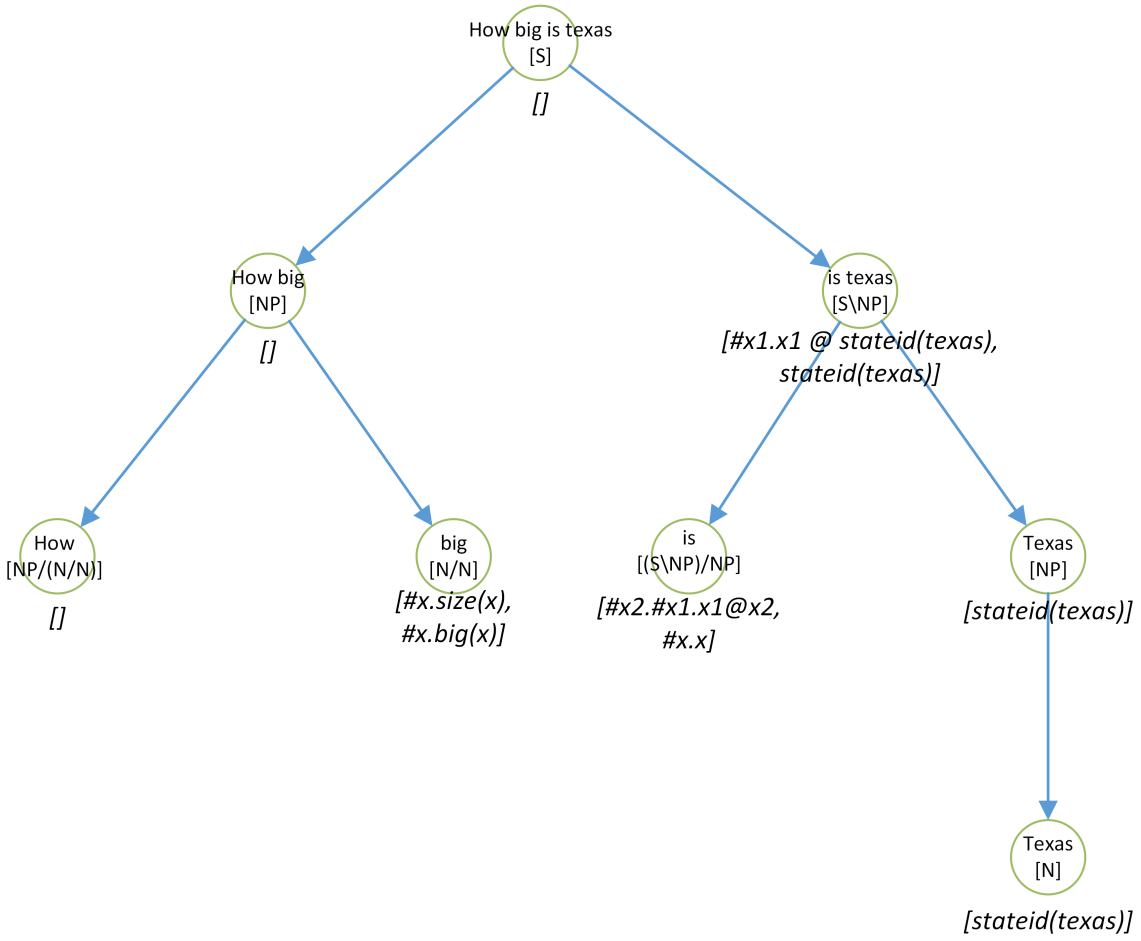

Figure 6.16: Current meanings in learning sentence ”How big is texas”.

tree, we then have to check each node to see if we can use inverse algorithm to learn new meanings. This naive approach can work in small examples, but it does not scale well.

Before introducing another approach, we need to define a few terminologies and principles that will be followed.

1. At any time in the learning process, we call the meanings queried from the lexicon *current* meanings (of words). These meanings can be from initial lexicon or learned in previous learning iterations and saved to lexicon. For example, *big* has two *current* meanings in Fig. 6.16.

2. Applying these meanings to the sentence/parse tree (like the one in Fig.6.15), we have the *current* meanings of phrases. For example, in Fig. 6.16, *is Texas* has two current meanings.

3. If one *current* meaning of the whole sentence is the same as its *given* meaning (given in the training set), we finish learning process.

4. We call the given meaning of the sentence the *expected* meaning.

5. Using this *expected* meaning (at the root of the tree) and *current* meanings of root's children, inverse algorithm gives us *expected* meaning of root's children.

6. If one node does not have any *expected* meaning, we do not learn further down that node.

7. We can repeat the process until we got *expected* meanings of the words (at leaf nodes). These *expected* meanings will be saved to the lexicon and will be used as *current* meanings in the next learning iteration.

Observing that two lambda expressions do not always combinable (i.e. $\sharp y.answer(size(y))$ to $stateid(texas)$), we propose a new learning algorithm. Instead of generating parse trees for each sentence - which can yield up to approximately k^n parse trees - we keep only one parse tree; and at each node, we keep two priority queues: one for *current* meanings and one for *expected* meanings. Each of these meanings is assigned one weight for ranking purposes. This new approach has a lot of advantages compared the previous ones:

- Although the worst-case complexity is still similar to the previous, the average case is better. Since two lambda expressions are not always combinable, if node A and B each have k *current* meanings, their parent has less than k^2 meanings. Thus, the whole sentence has much less than k^n *current* meanings on average.

- It takes much less time to compute and less space to store the new structure.
- It is easier to keep track of the new meanings (newly learned in current learning iteration) so that in each iteration, we can focus on learning and updating only new meanings.
- More importantly, we can assign weight for each meaning and prioritize learning on meanings with highest weights.

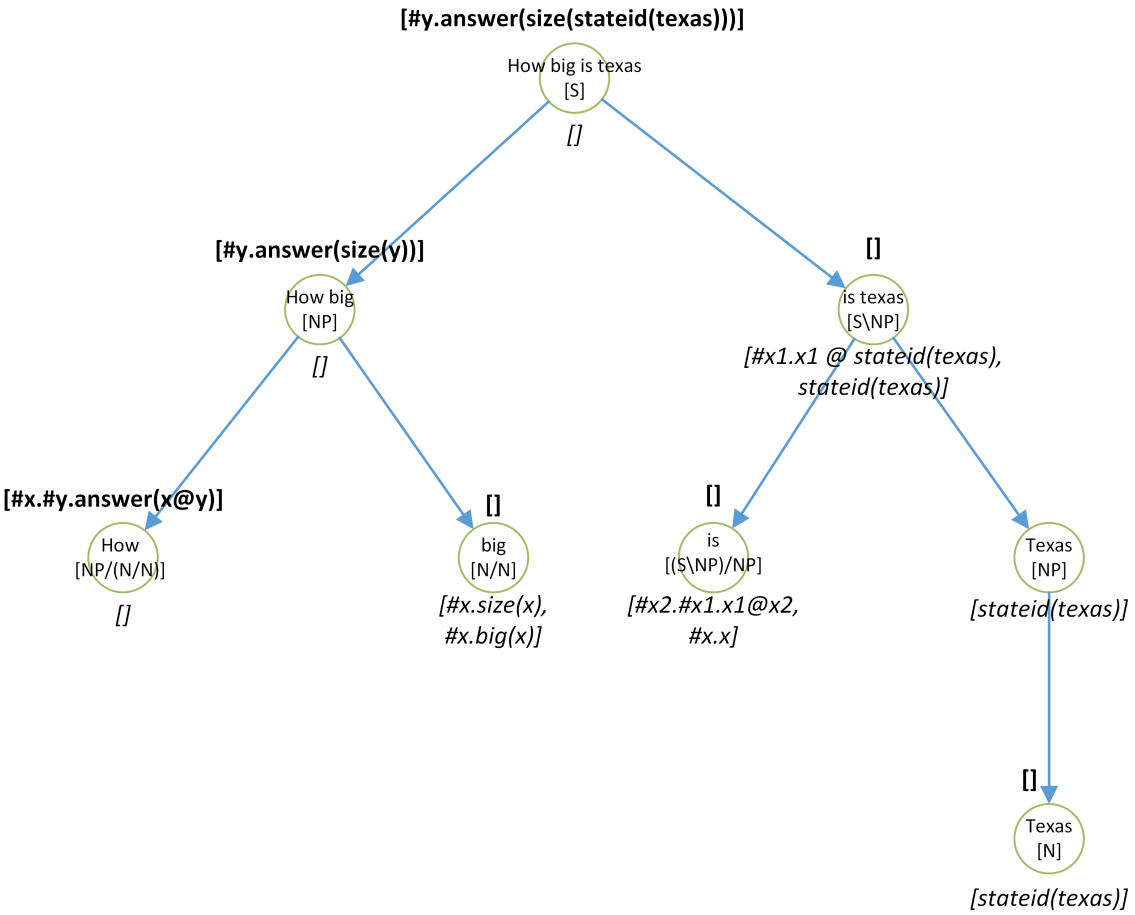

Figure 6.17: Expected meanings

The new learning process (named BottomUp-TopDown in Interactive Multistage Learning Algorithm below) is similar to that of the one in the previous example. First, the queues

of *current* meanings are built from bottom up (Fig. 6.16). Then, the queues of *expected* meanings are built top-down from the root. The difference is that one node can now have multiple *expected* meanings. Fig 6.17 shows the first step of the top-down learning process. The *expected* meaning of the whole sentence is $\sharp y.answer(size(stateid(texas)))$. Using this, we can learn the *expected* meaning of *How big*: $\sharp y.answer(size(y))$. Because *How big* does not have any *current* meaning, we cannot learn any *expected* meaning of *is Texas*. Learning on the branch of *is Texas* will be stopped due to the fact that there is no *expected* meaning to continue on. On the *How big* branch, *How* does not have any *current* meaning, so we have no *expected* meaning of *big*. Although *big* has two *current* meanings, only $\sharp x.size(x)$ can be used to compute *expected* meaning of *How*. The *expected* meaning $\sharp x.\sharp y.answer(x@y)$ of *How* is saved to the lexicon, preparing for the next iteration.

In the next learning iteration, *current* meanings are again built bottom-up (see Fig. 6.18). This time, *How* has one *current* meaning, making *How big*, and then *How big is Texas*, to have two *current* meanings. The first meaning of *How big is Texas* is indeed the *expected* meaning given in the training set; we have successfully learned this sentence and the learning process stops.

In the following, we present the Interactive Multistage Learning Algorithm (IMLA) following the idea we have presented in the example above. However, IMLA has a few more extra features:

1. When it cannot figure out the meanings of words using Inverse Lambda algorithm, it will attempt Generalization.

2. After a word is generalized, it will be generalized again to get more (possibly) new meanings once there is a relevant change in the lexicon. For example, if the word *Mississippi* is generalized from *Texas* to have meaning $stateid(Mississippi)$, it is kept on a waiting list. If we learn (by inverse) a new meaning of *Colorado* (same category

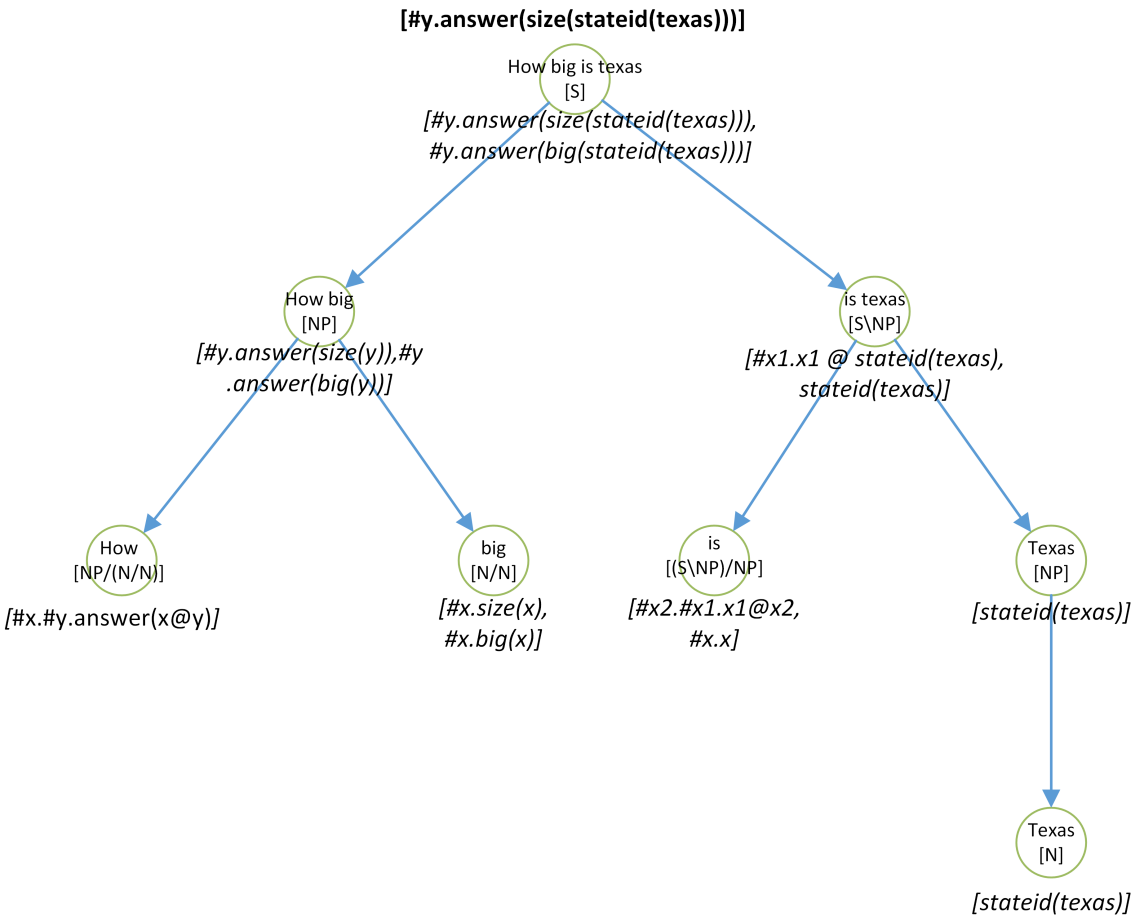

Figure 6.18: New iteration

NP as *Mississippi*), says $riverid(Colorado)$, *Mississippi* will be generalized again to get new meaning $riverid(Mississippi)$.

3. Each *current* and *expected* meaning can be *new* or *NOT new*. We keep track of and update this property accordingly so that at any learning iteration, we only concentrate on the new meanings and not all of the meanings we have learned thus far.

4. We assign weights for each meaning we obtain, like high weights for meanings in initial lexicon, high weights for meanings learned by inverse, and lower weights for the ones learned by generalization, for example.

Algorithm 1 IMLA algorithm

```
1: function IMLA(initLexicon,sentences, sentsMeanings)
2:     regWords ← ∅
3:     generalize ← false
4:     lexicon ← initLexicon
5:     repeat
6:         repeat
7:             repeat
8:                 for all s ∈ sentences do
9:                     newMeanings ← BT(s,lexicon,sentsMeanings)
10:                    lexicon ← lexicon ∪ newMeanings
11:                    for all n ∈ newMeanings do
12:                        ms ← GENERALIZE(regWords, n)
13:                        lexicon ← lexicon ∪ ms
14:                    end for
15:                end for
16:            until newMeanings = ∅
17:            if generalize=false then
18:                generalize ← true
19:                for all t ∈ unfinishedSents do
20:                    words ← GETALLWORDS(t)
21:                    ms ← GENERALIZE(words)
22:                    lexicon ← lexicon ∪ ms
23:                    regWords ← regWords ∪ words
24:                end for
25:            end if
26:        until newMeanings = ∅
27:        INTERATIVELEARNING
28:    until unfinishedSents = ∅ OR userBreak
29:    lexicon ← PARAMETERESTIMATION(lexicon,sentences)
30:    return lexicon
31: end function
```

Interactive Multistage Learning Algorithm (IMLA) (Algorithm 1) runs many iterations, each having multiple stages. In each iteration, (Stage 1) initially gets all sentences that we have not finished learning and tries the BottomUp-TopDown algorithm (Algorithm 2) to learn all possible meanings (by Inverse-λ). Each new meaning learned from this process is used to generalize the words on a waiting list. Initially, this waiting list is empty and will be added in stage 2. When no more new meanings are able to be learned through the BottomUp-TopDown algorithm, IMLA enters stage 2.

(Stage 2) IMLA gets all of the sentences for which the learning is not finished (unfinished sentences) and applies the generalization process on all of the words from those sentences. At the same time, it puts these words onto the waiting list, so that from now on, BottomUp-TopDown will try to generalize new meanings for them when it learns relevant meanings. After that, IMLA will return to stage 1. Next time, after exiting stage 1, it goes directly to stage 3.

(Stage 3) When both aforementioned stages cannot learn all of the sentences, the Iterative-Learning process is invoked, and all of the unfinished sentences are shown on the interactive GUI (Fig.6.19). Users can skip or provide more information on the GUI and the learning process is continued.

After finishing all stages, IMLA calls the ParameterEstimation algorithm to add a weight to each lexicon's tuple. Please see 6.4.3 for more details.

IMLA attempts to learn with the BottomUp-TopDown algorithm first, where Inverse-λ is used to get new meanings; this provides the highest quality meanings. When this fails, IMLA tries Generalization and then retries the BottomUp-TopDown algorithm again.

BottomUp-TopDown learning algorithm: For a given sentence, the CCG parser is used to get the CCG parse trees like the one of *how big is Texas* in Fig.6.19. For each parse tree, two main processes are called, namely bottom_up and top_down. In the first process, (Fig.6.3) all meanings of the words in the sentences are retrieved from the lexicon. These meanings are populated as leaves of a parse tree (see Algorithm 6.19). These meanings are combined in a bottom-up manner to have the meanings of phrases and the whole sentences. We call these meanings *current meanings*.

In the top-down process, using Inverse-λ algorithm, the given meaning of the entire sentence (called the *expected meaning* of the sentence) and the current meanings of the phrases, we calculate the expected meanings of the each phrase from the root of the tree to the leaves. For example, given the expected meaning of *how big is Texas* and current meaning of *how big*, we use Inverse-λ algorithm to get the meaning (expected) of *is Texas*.

This expected meaning is used alongside current meanings of *is* and/or *Texas* to calculate the expected meanings of *is* and/or *Texas*. The expected meanings of leaf nodes we have just learned will be saved to the lexicon and will be used in other sentences and in subsequent learning iteration. The top-down process is ceased when the expected meanings are same as the current meanings. Furthermore, in both the bottom-up and top-down processes, search beam algorithm is used to speed-up the learning processes.

Algorithm 2 BottomUp-TopDown (BT) algorithm

```
1: function BT(sentence,lexicon,sentsMeanings)
2:     parseTrees ← CCGPARSER(sentence)
3:     for all tree ∈ parseTrees do
4:         t ← BOTTOMUP(tree,lexicon)
5:         TOPDOWN(t,sentsMeanings)
6:     end for
7: end function
```

Interactive learning:

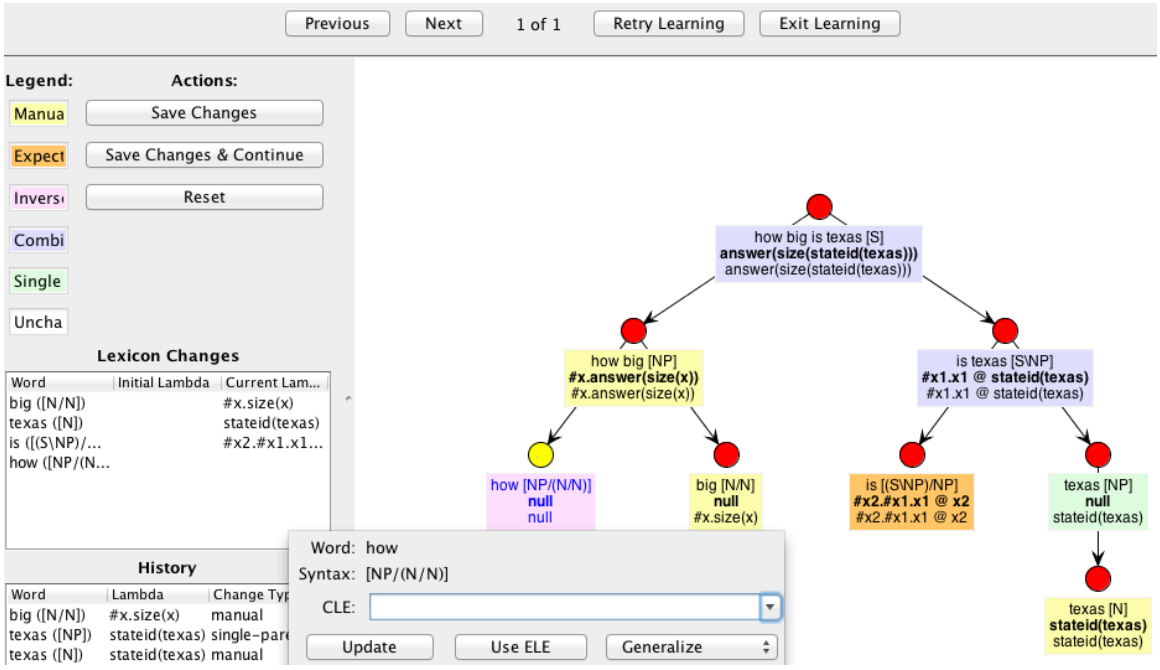

Figure 6.19: Interactive learning GUI. The boxes under each node show the corresponding phrases [CCG category], the expected meanings, and the current meanings. Clicking on the red node will show the window to change the current meaning (CLE).

In the interactive learning GUI, when users give additional meaning of word(s), the Lambda Application or Inverse-λ is automatically called to update the new meaning to related word(s). Once satisfied, users can switch back to the normal learning mode so that NL2KR can continue learning automatically.

Let us look at an example of how this process works. Let us consider the question "How big is Texas?" with meaning $answer(size(stateid(Texas)))$ (see Fig.6.19).

If, at the beginning, NL2KR has the meanings of all three words *how*-$\lambda x.\lambda y.answer(x@y)$, *big*-$\lambda x.size(x)$ and *Texas*-$stateid(Texas)$, in bottom-up process, meanings of *how* and *big* are combined to have the current meaning of *how big*-$\lambda x.answer(size(x))$. Since the meaning of *is* is unknown and we do not have current meaning of *is Texas*, in the top-down process, the expected meaning of *How big is Texas*-$answer(size(stateid(Texas)))$ is known; current meanings of *how big* is also known. Using them in the Inverse-λ algorithm, we can have the expected meanings of *is Texas*-$\lambda x1.x1@stateid(texas)$. Using this expected meaning and the current meaning *Texas*-$stateid(Texas)$, we can calculate the expected meaning of *is*-$\lambda x2.\lambda x1.x1@x2$. This newly learned expected meaning is saved into the lexicon. In the next iteration, the meaning of all the words in the question are known. Combining them all will give us the correct meaning of the entire question. Thus, the learning algorithm stops and Iterative-Learning is never called.

If initially, we have only two meanings, Texas-$stateid(Texas)$ and big-$\lambda x.size(x)$, NL2KR initially attempts to learn, but is not able to finish learning the whole question; it calls Iterative-Learning, which shows the interactive GUI (see Fig.6.19). If we provide that *how* means $\lambda x.\lambda y.answer(x@y)$, NL2KR will combine its meaning with *big* to have the meaning *how big*-$\lambda x.answer(size(x))$. It will then use Inverse-λ to figure out the meaning of *is Texas* and then the meaning of *is*. Now, all of the meanings are combined to have the current meaning $answer(size(stateid(Texas)))$ of *How big is Texas*. This meaning is the same as the expected meaning, so we know that the question is successfully learned. If users choose *Retry Learning*, the program will go back to IMLA, find that all of the sentences are learned, and then promptly exit.

6.4.3 Parameter Estimation

The Parameter Estimation module estimates a weight for each word-meaning pair such that the joint probability of the training sentences being translated to their given representation is maximized. It uses Probabilistic Combinatorial Categorical Grammars (PCCG) (Zettlemoyer and Collins, 2005, 2007, 2009).

Given a sentence S, its translation is M in $argmax_M\ P(M|S;\Theta)$ where $P(M|S;\Theta)$ is the

probability of translation of S to M, given the parameter vector. 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 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.

Unlike the previous works, here, we add another step of parameter estimation. Instead of using the default initial values of parameters, we first calculate these values (initial weights of each (word, CCG, meaning) tuple) based on number of times the corresponding tuples can be used to get the correct/incorrect meanings of the whole sentences. This step helps our parameter estimation process converge much faster.

6.4.4 Translation Algorithm

6.4.5 Translation

The goal of this module is to convert input sentences into the target formalism using the lexicon previously learned. The algorithm used in Translation module (Fig.3) is similar to the bottom-up process in learning algorithm but the Generalization is used first to guess the meanings of unseen words, and the validity of the output meanings is verified; then, they are ranked.

6.5 Experimental Evaluation

We have used NL2KR on various domains, namely Jobs, GeoQuery, and BioKR. The first two corpora are commonly used for the evaluation of sematic parsers. BioKR is a new corpus where the meaning of a sentence is represented in Answer Set Programming (henceforth, ASP) language, unlike Prolog, which is used in the other two corpora. Evaluation of NL2KR is done slightly differently from that of evaluation methods of other statistical translation systems. Rather than just reporting performance numbers for our system, our goal is to gauge the amount of effort compared to the return.

Algorithm 3 Translation algorithm

```
1:  function TRANSLATE(sentence, lexicon)
2:      candidates ← ∅
3:      parseTrees ← CCGPARSER(sentence)
4:      for all tree ∈ parseTrees do
5:          GENERALIZE(tree);
6:          t ← BOTTOMUP(tree)
7:          candidates ← candidates ∪ t
8:      end for
9:      output ← VERIFY-RANK(candidates)
10:     return output
11: end function
```

We compare the performance of our system for the GeoQuery and Jobs corpus with the following recently published, directly-comparable works, namely, the FUBL (Kwiatkowski *et al.*, 2011), UBL (Kwiatkowski *et al.*, 2010), λ-WASP (Wong and Mooney, 2007), ZC07 (Zettlemoyer and Collins, 2007), and ZC05 (Zettlemoyer and Collins, 2005) systems. For both of the corpora, we use the same setup as FUBL, ZC07, and ZC05. The BioKR corpus initial experimentation has been done using 10 fold cross-validation.

6.5.1 Corpora

GeoQuery

GeoQuery (Zelle and Mooney, 1996) is a corpus containing questions which can be executed against a database of the geographical information of the United States. It contains a total of 880 questions their meanings in a Prolog. We follow the standard training/testing split of 600/280. An example sentence meaning pair is shown below.

```
Question: How long is the Colorado river?
Meaning : answer(A,(len(B,A),const(B,riverid(colorado)),river(B)))
```

Jobs

The Jobs (Zelle and Mooney, 1996) dataset contains a total of 641 job related queries and their translations in Prolog. The queries can be directly executed against a database of job listings, where each of them specifies a list of criteria for searching jobs. An example sentence meaning pair from the corpus is shown below.

```
Question: What jobs are there for programmers that know assembly?
Meaning : answer(J, (job(J),title(J,T),const(T,'Programmer'),
    language(J,L),const(L,'assembly'))))
```

BioKR Corpus

BioKR is a corpus of deep reasoning "How" biology questions, collected from a the Biology Questions and Answers Site [3] for testing a reasoning-based QA system. The required target language is Answer Set Programming(ASP) (Gelfond and Lifschitz, 1988) but since the ASP format is long and clumsy, instead of converting to it directly, we use a more concise intermediate language which can be directly converted into ASP (Table 6.7 Col 3.) [4]

Recognizing and correctly parsing biological entities is a challenge for parsers trained on usual English text. This problem is usually resolved by preprocessing the corpus to replace all biological entities by place-holders. Our goal was to evaluate the translation process in isolation, without being affected by an Entity Tagger's perfomance. To do this, we identified the substrings that were arguments of any predicates in the formal representation and replaced them by place-holders. For e.g. the sentence, "How does vitamin C act in the body?" with translation"r2('vitamin C', work) $\wedge$ scope(body) $\wedge$ type(how)" was replaced by the sentence, "How does ent0 act in the ent1?". An example of a preprocessed sentence in the corpus is shown in Table 6.7 Column 4. This corpus consisted of more than 100 manually annotated questions.

6.5.2 Initial Dictionary Formulation

GeoQuery First of all, we select a set of 100 structurally different sentences from the training set and bootstrap the learning process with an initial dictionary. This dictionary contains nouns for Geoquery entities (such as states, rivers, or cities, etc) and question

[3] http://www.biology-questions-and-answers.com

[4] Note that $r3(x,y,z)$ is just a generic relation which means that x and z have the relation type y.

Table 6.7: BioKR Corpus

English	Formal Lang.(ASP)	Intermediate Lang.	Preprocessed
How are sunlight and sugar related in photosynthesis	question(q1). has(q1, type, how). has(q1, category, relation). has(q1, param1, sunlight). has(q1, param2, sugar). has(q1, scope, photosynthesis)	r3(sunlight, related sugar) ∧ type(how) ∧ scope(photosynthesis)	r3(ent0, related,ent1) ∧ type(how) ∧ scope(ent2)

words. The meanings of those words were easy to obtain; they follow simple patterns. We then train the translation system on those selected sentences. Because that dictionary is not enough for learning all the sentences, the learning process stops and shows the interactive learning GUI, asking for further meanings. 45 more meanings were provided with the help of the interactive learning GUI, increasing the total meanings initially given to 119 (119 word meanings tuples (Fig. 6.20, #<*word, category, meaning* >). From this, the NL2KR system learned 1793 tuples, which clearly shows that the amount of initial effort is substantially less compared to the return. These numbers illustrate the usefulness of the NL2KR GUI, as well as the NL2KR learning component. One of our future goals is to further automate the process and reduce the GUI interaction part.

Figure 6.20 shows the size of the learned dictionary, the initial dictionary, and the number of meanings humans need to derive using interactive GUI on three bases: (1) number of unique meanings across all the entries in the dictionary, (2) number of unique <*word, category, meaning*> entries, (3) and number of unique <*word, category*> pairs. Please note that the reported initial dictionary includes the GUI driven meanings.

The total number of unique <*word, category*> pairs in the training corpus provides a lower bound on the size of the ideal output dictionary, since each unique <word, CCG category> pair must have at least one meaning.

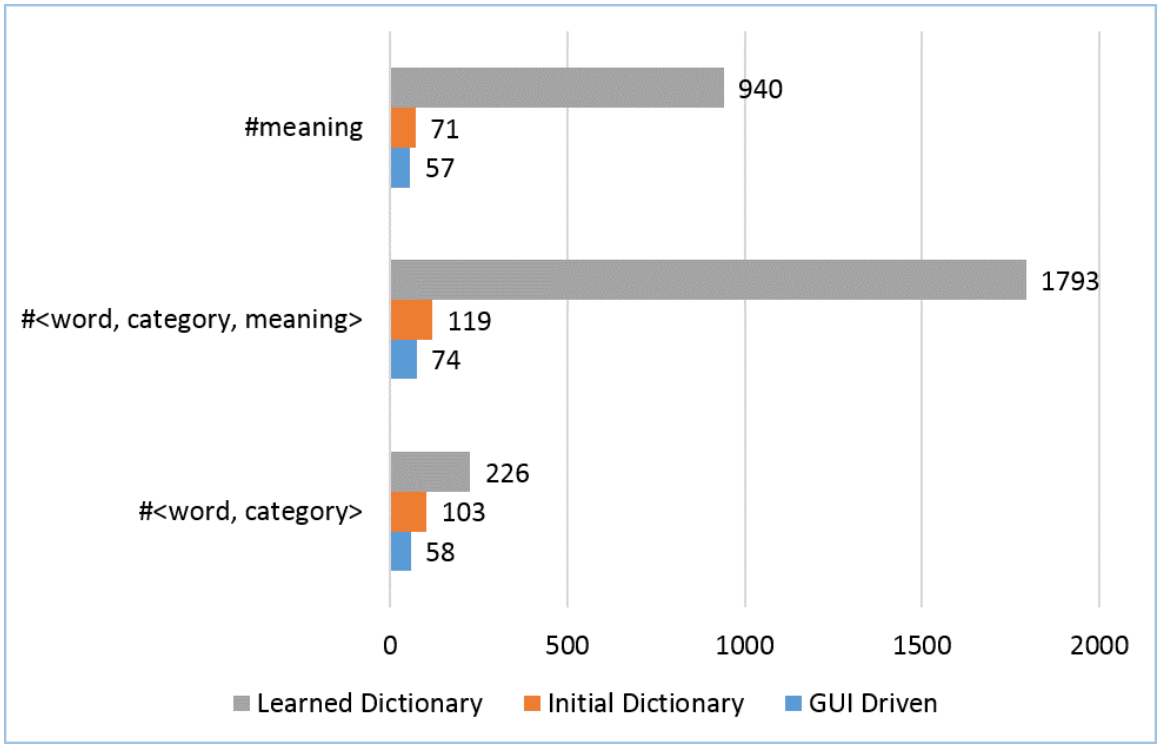

Figure 6.20: Comparison of Initial and Learned dictionary for GeoQuery corpus on the basis of the number of entries in the dictionary, number of unique <word, CCG category> pairs and the number of unique meanings across all the entries. "GUI Driven" denotes the amount of the total meanings given through interactive GUI and is a subset of the Initial dictionary.

There were many words, such as "of" and "in," that had multiple meanings for the same CCG category. So one *<word, category>* may have multiple meanings. The ideal dictionary can be much bigger than the lower bound. The learned dictionary contained 27 and 42 different CCG categories for GeoQuery and Jobs corpora respectively. The minimum number of unique meanings for CCG category was 1 for both of them, whereas the maximum went to 149 and 132 respectively.

We also evaluate the total effort one must spend for the experiments. To perform the experimentations on GeoQuery corpus, a student, who had no prior knowledge about the

system and CCG formalism, took 40 hours. However, he took around 20 hours for the next experiment on Jobs corpus. Out of these 60 hours, a total of 3 hours were spent in interactive learning mode. NL2KR expects users to know about Lambda application and the six combinators of CCG.

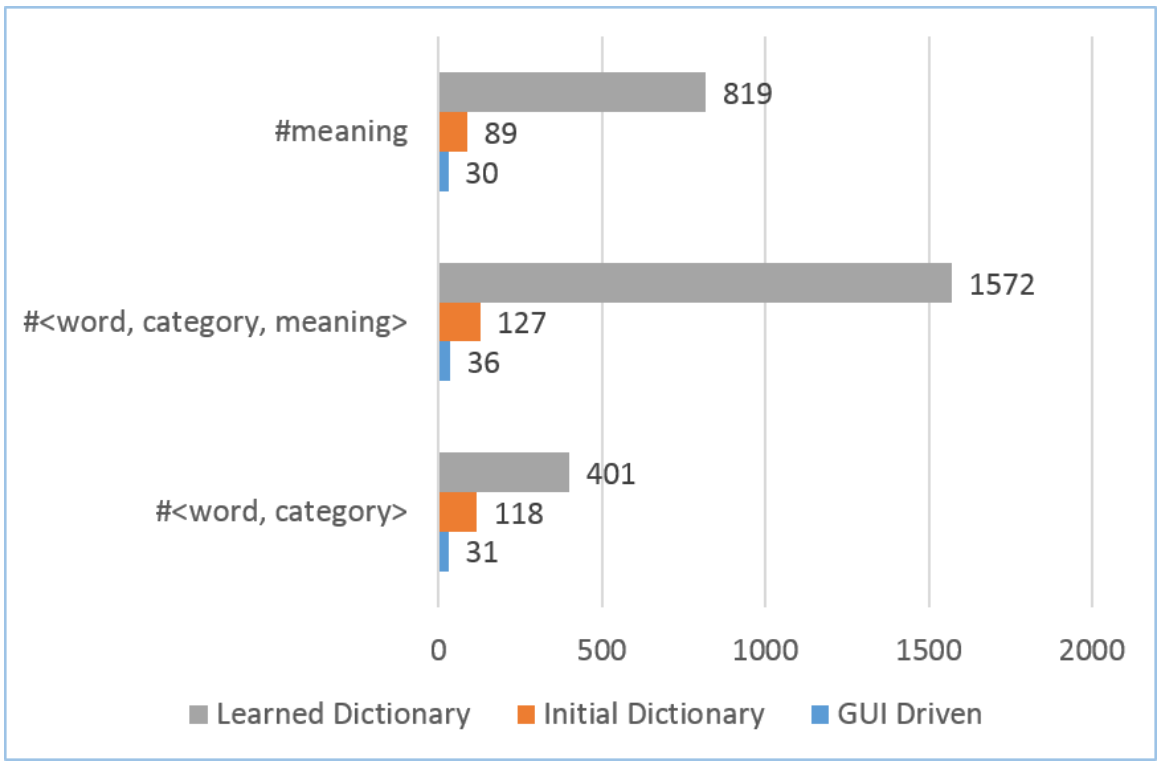

Figure 6.21: Comparison of Initial and Learned dictionary for Jobs corpus.

Jobs For the Jobs dataset, we followed a similar process as that of the GeoQuery dataset. 120 structurally different sentences were selected, and an initial dictionary of size 127 was constructed. From them, NL2KR yields a learned dictionary of size 1,572. Fig. 6.21 compares the initial and learned dictionary for Jobs. Again, we can see that the amount of initial effort is substantially less in comparison to the return.

System	Recall	Precision	F1
ZC05	79.3	96.3	87.0
ZC07	86.1	91.6	88.8
λ-WASP	86.59	91.95	89.19
UBL	87.9	88.5	88.2
FUBL	88.6	88.6	88.6
NL2KR	92.1	91.1	91.6

Table 6.8: Geo880

System	Recall	Precision	F1
UBL	81.8	83.5	82.6
FUBL	83.7	83.7	83.7
λ-WASP	75.6	91.8	82.9
NL2KR	92.4	81.0	86.3

Table 6.9: Geo250

System	Recall	Precision	F1
ZC05	79.3	96.3	87.0
ZC07	86.1	91.6	88.8
λ-WASP	86.59	91.95	89.19
UBL	87.9	88.5	88.2
FUBL	88.6	88.6	88.6
NL2KR	94.03	95.43	94.01

Table 6.10: Jobs640

System	Recall	Precision	F1
NL2KR	93.0	98.0	95.0

Table 6.11: BioKR

6.5.3 Results and Discussion

Tables 6.8 , 6.9, and 6.10 present the comparison of the performances of NL2KR on the GeoQuery and Jobs domain with other recent works. NL2KR obtained a 91.1% precision value, 92.1% recall value, and a F1-measure of 91.6% on GeoQuery (Fig. 6.22, Geo880) dataset. For Jobs corpus, the precision, recall, and F1-measure were 95.43%, 94.03%, and 94.01% respectively. Figure 6.22 shows those performances on a precision-recall space. We draw the F1 curves (all points on a curve have same F1 score) in the figure to illustrate the differences in F1-score of the systems.

In all cases, NL2KR achieves state-of-the-art recall and F1 measures, and it significantly outperforms FUBL on GeoQuery. Table 6.11 shows the results for BioKR .

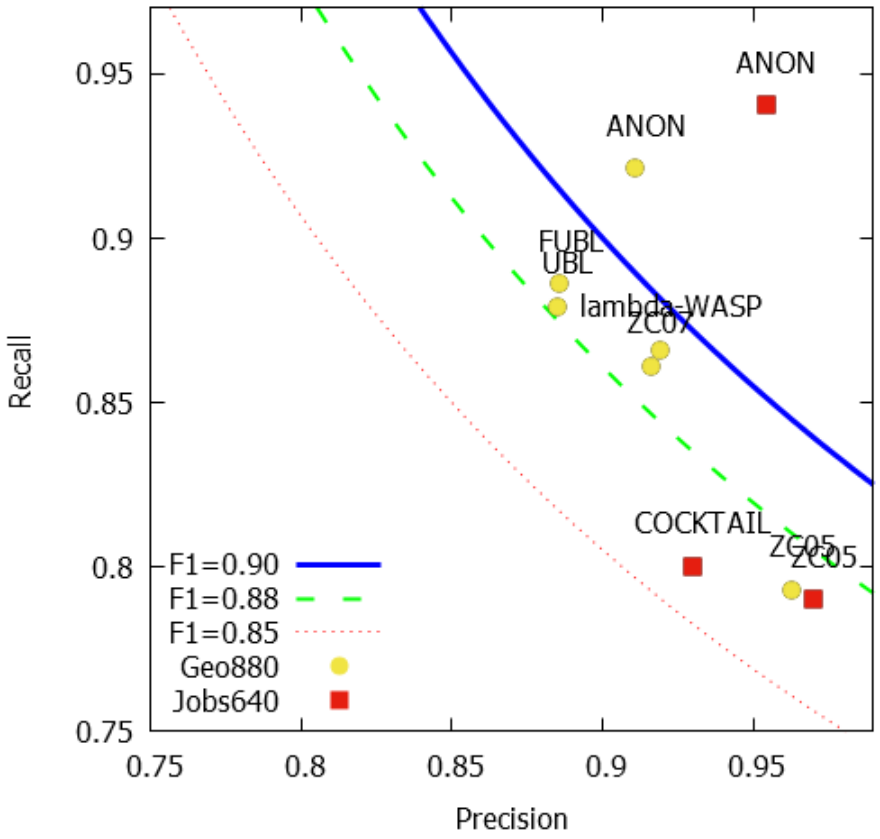

Figure 6.22: Comparison of Precision, Recall and F1-measure on GeoQuery and Jobs dataset.

We believe NL2KR's high recall is due to the higher quality meanings learned by NL2KR. These meanings are more general and reusable; meanings learned from one sentence are more likely to be applied again in the context of other sentences. Another key factor is the consistency of our CCG parser, which allows Generalization to work more effectively in the process of translation. By consistency, words' CCG categories and CCG parse trees are similar in sentences, having similar syntactic structures.

Analyzing sentences that NL2KR was not be able to translate, we found that the culprit is their structural differences. Their structures are not identical to any of the sentences present in the training dataset, or they could not be constructed by combining word/CCG

category pairs observed in the training sentences.

For sentences in which NL2KR gives the wrong translation, we found that NL2KR usually has a correct meanings as one of the possible meanings. However, the correct meanings were not the picked because their weights were not the top ones. It may be noted that although our precision is lower than that of ZC05, ZC07, and WASP, we have achieved significantly higher F1 measures than those of the others. For BioKR corpus, the number of meanings for each word is lower than GeoQuery corpus, which contributes to the higher precision in BioKR corpus.

6.6 Take Out Lessons

From our experience in developing and doing experiments with NL2KR, we have learned some lessons (about how to use NL2KR more effectively) that we would like to share with other users.

6.6.1 What Is a Good Corpus?

A good training set should have sentences ranging from easy to hard. This would help NL2KR gradually enlarge the dictionary and require less input from user.

6.6.2 What Is Structure the Target Language Should Be?

Some target languages are not optimal for approaches like NL2KR. For example, consider one translation in Prolog from Jobs640 corpus.

```
answer(J, job(J),language(J,L), const(L,'java'))
```

Since J does not need to be fixed, we can replace J by X, but still have correct meanings. In order to let NL2KR capture this property, we need to represent translation with $\sharp x.answer(x, job(x), language(x,l), const(l,' java')))$. This way, when the translation is $\sharp y.answer(y, job(y), language(y,l), const(l,' java')))$, NL2KR can still recognize the equivalence.

Suppose the meaning of the question is
$answer(X, count(B, state(B) \wedge loc(B, const(O, countryid(usa))), X))$. As there are appearances of variables X, B, and O, the meaning of words in the question must have those variables. The quality of the translation system trained by NL2KR will depend on the quality of the initial dictionary. If we slightly, "arbitrarily" split the meaning, most of the time, one word's meaning (i.e. of "state") will contain variable "B" and other words' meanings (i.e. "in") will also contain "B." Although the system can successfully learn the meaning of each word in question, those meanings are very specific to this question and cannot be used for similar sentences. In other words, the trained system more or less "remembers" how to translate this question and is likely to not perform well with different questions.

The proper way to proceed, in our experience, is to use λ variables like x in
$\sharp x.answer(x, job(x), language(x, l), const(l, 'java')))$. So for
$\sharp x.answer(x, job(x), language(x, l), const(l, 'java')))$, we also need to either rewrite to eliminate variable l or introduce λl to have something that looks like
$\sharp x.\sharp l.answer(x, job(x), language(x, l), const(l, 'java')))$.

6.6.3 How to Construct the Initial Lexicon

Constructing a good initial lexicon plays a crucial role in building a translation system. For most users, we recommend to follow the guidelines below.

1. Pick a small number of sentences; start with easy ones.

Example

- how big is texas : $answer(size(stateid(texas)))$
- how large is texas : $answer(size(stateid(texas)))$
- where is dallas : $answer(X, loc(cityid(dallas), X))$
- what is the area of texas : $answer(area(stateid(texas)))$

2. Observe and find the common patterns:

- Texas = $stated(Texas)$
- Dallas = $cityid(Dallas)$
- Big $\sim$ size $\sim \lambda x.size(x)$
- Area $\sim$ area $\sim \lambda x.area(x)$
- Is $\sim$ no specific meaning, probably has $\lambda x.x$ meaning.
- The $\sim$ no specific meaning, probably has $\lambda x.x$ meaning.
- Of $\sim$ no specific meaning, probably has $\lambda x.x$ meaning.
- How $\sim$ answer( )
- Where $\sim$ answer(X, loc(,X))
- What $\sim$ answer( )

3. Build the initial dictionary, starting with simple sentences, adding one by one. Use the iterative GUI. Can start with empty initial dictionary.

4. Pay attention to the CCG parse trees - which is the argument, which is the function of the lambda application.

5. Give the meaning that we are most confident of; try to use the interactive GUI to figure out the others.

6. Skip the hard sentences until we gain more experience.

7. Update the initial dictionary after we successfully learned some sentences.

6.6.4 *Patterns for λ Expression*

We also include in the NL2KR's document many tricky and difficult patterns for λ expressions that users can employ in their application. For example, constructing G' from G: $G' = \lambda x.x@G$ gives us the property: $G'@F = F@G$. This technique, called flipping, is very useful when the desired direction of λ-application is different from the one pointed by CCG parse tree.

6.7 Related Work

Montague's approach has been widely used in (Blackburn and Bos, 2005; Zettlemoyer and Collins, 2005; Costantini and Paolucci, 2010; Baral *et al.*, 2011; Kwiatkowski *et al.*, 2010, 2013; Tang and Mooney, 2001; Ge and Mooney, 2005; Kate and Mooney, 2006; Wong and Mooney, 2006, 2007) to translate natural language to formal languages.

We now compare our system with (Zettlemoyer and Collins, 2005) (henceforth, ZC05) and (Kwiatkowski *et al.*, 2010)(henceforth, KZGS10), which are the most similar to our work in that they also use Montague's approach for converting natural language sentences to logical form and uses CCG formalism to derive the syntactic mode of combination of the constituents present in the sentence. However, the learning algorithms in ZC05 (henceforth, Template Based GENLEX) and in KZGS10 (henceforth, Unification Based GENLEX) are significantly different from IMLA.

Template Based GENLEX algorithm, along with an initial dictionary, requires the user to provide the semantic templates for all words. A semantic template is a λ-expression (e.g. $\lambda x.p(x)$ for an arity one predicate), which describes a particular pattern of representation in that formal language. From these templates, the learning algorithm extracts the semantic representation of the words from the formal representation of a sentence. Entries in lexicon are created by pairing each possible substring in the natural language sentence with the extracted meanings, and ranking the associations according to some goodness measure.

However, manually coming up with semantic templates may not be a good idea, since it is laborious and requires deep understanding of translation to the target language.

UBL (Kwiatkowski *et al.*, 2010) learns the meanings in a brute force manner, restricting the choices of formal representation. Given a sentence S and its representation M in the restricted formal language, it breaks S into two smaller substrings $S1, S2$; M into two λ-terms $M1, M2$ (which combine to produce M) using an algorithm called higher-order unification.

It then considers the meanings of the words/phrases as the pairs $< S1, M1 >$ and $< S2, M2 >$, and it recursively learns by breaking $S1, M1, S2, M2$ further. Because there are many ways to break S as well as M, UBL needs to consider all possible splittings and their combinations, "outputs a large number of entries" (Artzi *et al.*, 2013) (word-meaning pairs) which will obviously limit the scalability. More importantly, the algorithm to split M imposes various restrictions which severely limits its applicability to new applications. For example, it limits the number of conjunctions in a sentence, and forms of functional application on the meaning represent language.

Consider a sentence "I want a flight to New York" with a meaning "h". UBL first finds all possible substring pairs such as

- "I" vs. "want a flight to New York"
- "I want" vs. "a flight to New York"
- ...
- "I want a flight to" vs. "New York"
- "I want a flight to New" vs. "York"

For each split, the corresponding CCG categories are assigned to each substring of the pairs. The meaning "h" then also be split into to pairs of f and g such that $h = f(g)$ or $h = \lambda x.f(g(x))$. For one particular mapping between two pairs, the process is repeated

until all words are processed. Doing it this way, the CCG parse tree structure generated is not necessarily making sense or "equivalent" to English grammar, or in author's words (Artzi *et al.*, 2013), the "parse structure is latent".

Though similar in spirit, NL2KR takes a different angle on the learning algorithm and formulates the learning problem as a dynamic programming problem. Instead of inducing CCG categories, our parser integrates information from state-of-the-art parsers like the Stanford parser (Socher *et al.*, 2013) to get linguistically robust parse trees.

For example, for the sentence "I want a flight to New York," we will first get the CCG parse tree. It then uses Inverse and Generalization algorithms to proceed in a "bottom-up then top-down" fashion. It begins with meanings of words from the initial dictionary and those obtained by Generalization, and uses the Inverse λ algorithm to find the missing meanings of phrases/words.

For example, if meanings of *I* is known, inverse lambda algorithm will be used to learn the meaning of *want a flight to New York* from a given meaning of the whole sentence. The process is continued on the *want a flight to New York* branch. Meanings of *want* or *a flight to New York* are used to learn the meanings of the other. The process stops when we cannot learn anything more. Please see section 6.4.2 for more details.

Since Inverse Algorithm finds exact meanings of a phrase or a word, the output dictionary is compact, which would help the system scale efficiently.

Moreover, this our allows us to keep CCG categories and semantics consistent with intuition, thereby making the system more generic; given a new unseen sentence, it is more likely to get the correct meaning. For example, assume that we already trained a translation system and have an output lexicon. With a new sentence, which may have some words that have not appeared in the training corpus, and thus are not in the output lexicon, the stable CCG parser increases the chance to have the correct CCG categories of those words, however. Since these CCG categories are used to look up other words in the lexicon for

generalization, the chance of generalizing correct meanings is also increased.

Also, NL2KR does not impose any restrictions on the representation. The current learning algorithm uses a seed lexicon to bootstrap the learning process. Our approach allows us to generate fairly complex λ-expressions if needed while maintaining good training speed, without keeping restrictions like limiting the number of conjunctions (Kwiatkowski *et al.*, 2010).

6.8 Knowledge Parser

There has been a lot of advancement in the field of Natural Language Understanding since the first attempt by Daniel Bobrow in form of the program STUDENT in 1964, which was developed as an attempt to read and solve algebra problems of high school level (Bobrow, 1964; Russell *et al.*, 1995; McCorduck, 2004). It was followed by systems like, ELIZA in 1965, an interactive dialog system in English (Weizenbaum, 1976).

At that point of time researchers started focusing on the representation of text, which is supported by introduction of Conceptual Dependency theory by Roger Shank in 1969 (Schank and Tesler, 1969), Augmented transition Network by William A. Woods in 1970 (Woods, 1970) and further advancements by Terry Winograd (Chrisley and Begeer, 2000). Attempts were made towards Processing Natural Language (Chrisley and Begeer, 2000) after that and a new level of intelligence was achieved after every attempt.

Project Halo is a long-term research program of Vulcan Inc to develop "Digital Aristotle" (Friedland *et al.*, 2004), a program to understand and reason over various scientific disciplines. Under project Halo, in 2004, Automated User-centered Reasoning and Acquisition System (AURA) was reported to be the first program which can archive remarkable scores in Advanced Placement exams in Biology, Chemistry and Physics. In later phase, "Inquire Biology" introduced in 2013 (Chaudhri *et al.*, 2013) as an intelligent book application on tablet. It is an enhancement of the popular Campbell Biology book which can

answer various questions posed by students.

The knowledge bases in AURA were manually constructed by experts in the fields, which were very expensive and time consuming. Automatically construct such knowledge bases would have huge impact on not only Natural Language Understanding but also many other fields. Imagine how such knowledge base on common sense knowledge is helpful to planning, predict actions from video footage or to digital assistant like Siri or Google Now.

However, automatically constructing such knowledge bases, presumably from text, is a tough challenge that required years to be conquer as shown in its first step, representing natural language sentences. In (Chaudhri *et al.*, 2011), Chaudhri pointed out representing a natural language sentences need more knowledge than what the sentence explicitly contains as the analogy to an iceberg in Figure 7.2. Vocabulary knowledge and Explicitly Stated Knowledge are shown at the tip of the iceberg. These respectively are the words in the sentence and the sentence's logical restatement. Hidden underwater are Linguistic and Background (Core) knowledge, which are the knowledge needed to really understand the sentence.

With this motivation in mind, we developed the initial version of a Knowledge Parser (Sharma *et al.*, 2015b,a) which convert sentences in natural language to a representation which is useful for question answering, reasoning, etc. For representation, we start from Knowledge Description Graphs and Component Library in KM (Clark and Porter, 2011) and adding more vocabularies to handle logical symbols, values, conditions, etc.

Most of the current semantics parsers and role labeler concentrate on the first and second layers in Figure 7.2. Some recent systems also provide Linguistic Knowledge (3rd layer) from FrameNet, VerbNet, PropBank, etc. Since each system has its own strength and its own format, which are not really compatible to the others, our goal is not building another semantics parser but leveraging from existing systems. We want to take information from many sources, combining them using a rich vocabulary ontology, searching and

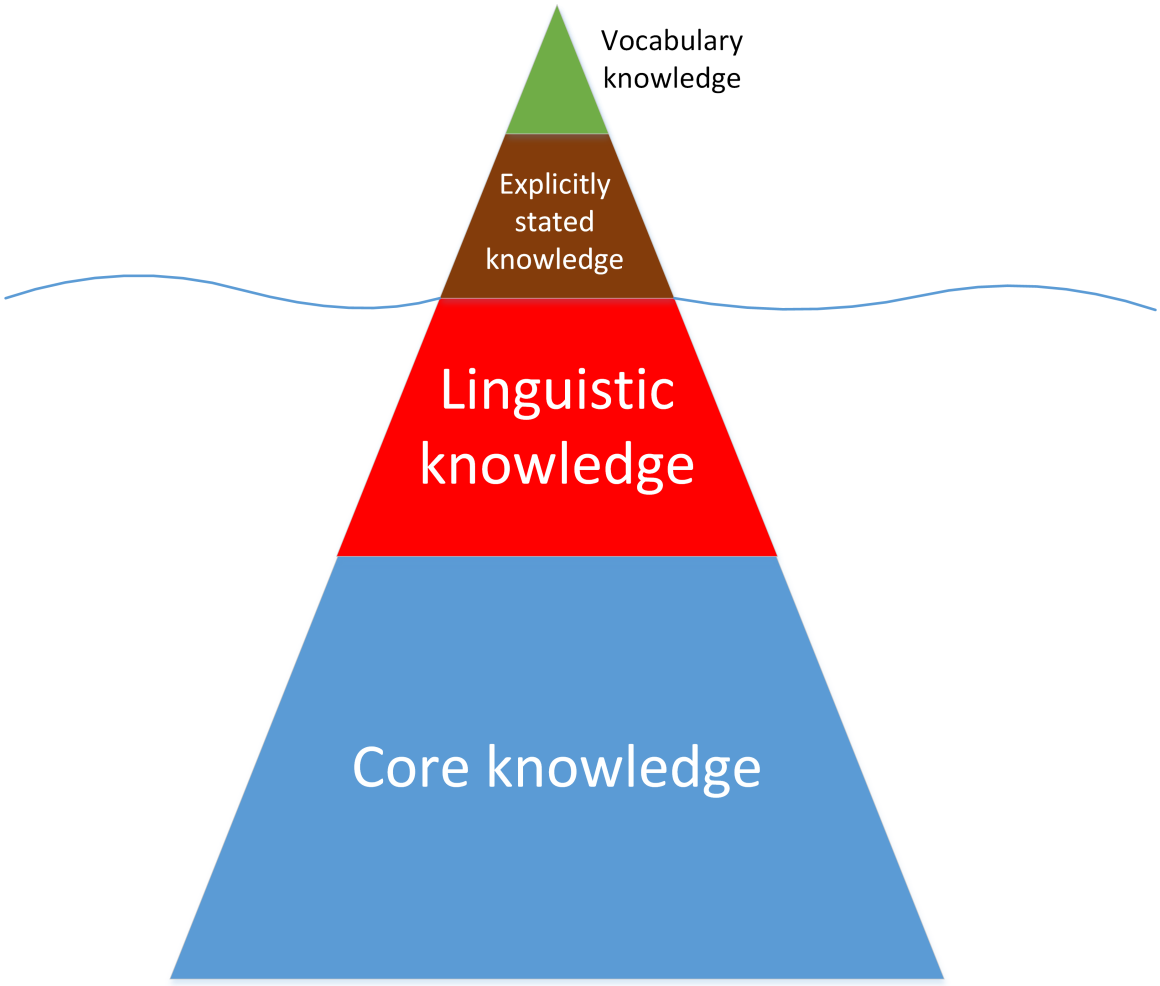

Figure 6.23: Iceberg of knowledge needed to represent a natural language sentence.

Vocabulary Knowledge: class, relation and function symbols representing each word (phrase) in the sentence.

Explicitly Stated Knowledge: logical representation of the sentence using the vocabulary knowledge.

Linguistic Knowledge: English knowledge required to understand the sentence.

Core Knowledge (Background Knowledge): all other knowledge needed to understand the sentence.

obtaining extra information if needed, to provide the knowledge in all four layers from Vocabulary to Background knowledge (Figure 7.2).

Details of how Knowledge Parser was implemented can be found in the two papers: Sharma *et al.* (2015b,a).

6.9 Conclusion

We presented the NL2KR system, which is a DIY kit to build translation systems that convert sentences from natural language to their equivalent formal representations in a wide variety of domains. We described the system algorithms and architecture and demonstrated its effectiveness in GeoQuery, BioKR, and Jobs domains. The system also includes a CCG parser, which can be used as a standalone parser and has certain advantages over its predecessors.

Chapter 7

CONCLUSION AND DISCUSSION

7.1 Conclusion

In this dissertation, we developed a system to answering deep questions that goes beyond simple phrase matching in text. We defined and developed several modules toward this challenge.

Initially, we defined the notion of Knowledge Description Graph (KDG), a graphical structure containing information concerning events, entities, and classes. We proposed formulations and algorithms to construct KDGs from a frame-based knowledge base.

We then defined the answers of various "How" and "Why" questions with respect to KDGs and discussed deriving missing information when constructing KDGs from under-specified knowledge bases. Moreover, we suggested how to obtain the answers from KDGs using Answer Set Programming, and also how to answer many factual question types with respect to the knowledge base.

Next, we considered how to construct the KDGs. We investigated two directions: (i) from frame-based knowledge bases and (ii) from natural language text. In the frame based knowledge bases that we investiaged, information from one instance is not only listed in its classes but also buried in those of the ancestors and related classes. To get the complete information, one must go through a process called "unification" to combine and unify the information from all sources. In the process, redundant or conflicted data must be resolved. Toward this goal, we proposed algorithms and formulations to solve the unification in a recursive manner.

Towards the goal of building KDGs from natural language text, we developed the

NL2KR system and Knowledge Parser. Knowledge Parser helps in converting natural language to KDG specification, while NL2KR system helps in building translation systems from natural language to formal language. The target languages are not solely tied to KDG specification language, but rather can be in a wide range of formal language. NL2KR's usages, thus, are not limited to building KDGs but can be extended to many more translation tasks: robot control, puzzle solving, etc. In this dissertation, we used NL2KR to convert deep queries in natural language to one formal representation that is suitable for reasoning (please see Chapter 6 for more details).

7.2 A Path to Natural Language Understanding

Regardless of converting natural language to KDGs, a parser called Knowledge Parser (Sharma *et al.*, 2015b,a) has been under development (not included in this dissertation). Knowledge Parser takes information from many sources, creating a graphical semantic representation of the input text like KDGs. With all of those modules, we are completing one path to Natural Language Understanding (NLU).

Shown in Figure 7.1 is our suggested path to NLU, which centers around KDGs. Facts from natural language text, knowledge bases, and prior knowledge all are converted to KDGs, which can unify, enrich, and complete missing information as shown previously. From there, we can answer factual questions, comparison question, and deep questions. Moreover, we believe our approach can be used to solve a wide range of applications in NLU, including (but not limited to):

1. Answering deep reasoning questions like "Why" and "How" questions
2. Solving reading comprehensions
3. Coreference Resolution
4. Machine translation

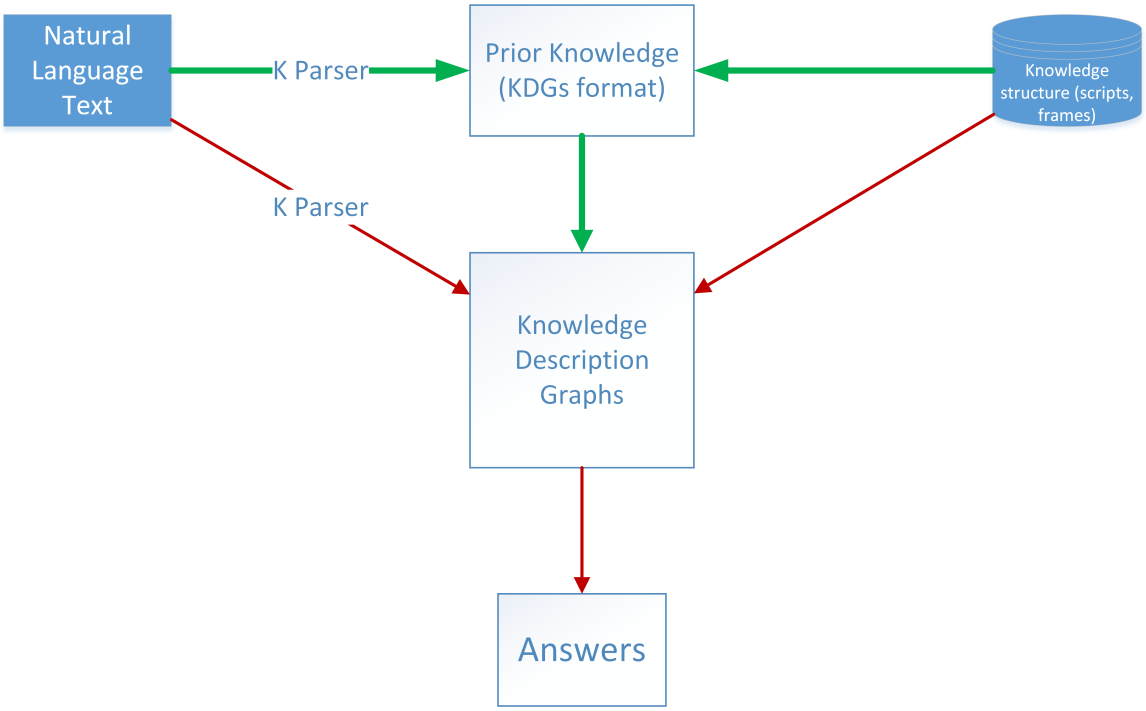

Figure 7.1: A path to Natural Language Understanding.

5. Building common sense knowledge bases like ConceptNet but with richer information.

6. Solving textual entailment problems

7. Making the computer understand text and reason the way humans do

8. Intelligent personal assistants like Siri

Among those applications, besides answering deep questions, we have been investigating Coreference Resolution like Winograd Schema Challenge (Levesque *et al.*, 2012). Winograd Schema Challenge corpus contains pairs of sentences and questions, i.e. "The man could not lift his son because he was too heavy. Who was heavy?", that require human-like reasoning to answer. In (Sharma *et al.*, 2015c,b), we showed that many cases in Winograd Schema Challenge can be solved by converting input sentences to KDGs, searching

suitable background knowledge while converting to KDGs, and comparing and reasoning those KDGs to draw conclusions (such as “he” means “the man” in previous example). In the future, we will tackle the applications mentioned above, such as Reading Comprehension, building knowledge bases like ConceptNet. While doing that, as needed, we will continue to improve various modules, like Knowledge Parser or Knowledge Hunting.

7.3 Toward a Complete Deep Question Answering System

7.3.1 Deep Question Categorization

In the collection of 1,839 questions and answers [1] , there are 265 ”How” questions and 89 ”Why” questions. Analyzing “How”questions, we found out:

1. 18 questions are actually factual questions, not deep, because they are “How long?” and “How old?” For example, “How long is the incubation period of the HIV?”

2. 11 questions are closer to “Why” than ”How.” For example, “How can the fact that fish and dolphins have similar organs and general shapes be explained?”

3. 9 questions are “How are they related” questions. i.e. “How are the concepts of chromosome, chromatin, and chromatids related?”. These questions are discussed in section 2.7.2.

4. 21 are “How different” questions; “How different are oxyhemoglobin and hemoglobin?” The answering of these questions is discussed in section 3.7.2.

5. Approximately less than 20 questions do not directly match our “How” pattern. e.g.

   (a) How to find the number of pairs of alleles involved in polygenic inheritance using the number of phenotypical forms of the traits they condition.

[1] http://www.biology-questions-and-answers.com/

(b) How is it possible to obtain the probability of emergence of a given genotype formed of more than one pair of different alleles with independent segregation from the knowledge of the parental genotypes?

(c) How can the hypothesis that asserts that chloroplasts and mitochondria were primitive prokaryotes that associate in mutualism with primitive anaerobic eukaryotic cells be corroborated?

While it is not impossible to cast many of these questions to our known templates, we still do not consider them matched because of several reasons. First, they ask for more "abstract" knowledge, such as general knowledge about experiments or data processing on top of the biological knowledge. Second, this "abstract" knowledge is less likely to be encoded in the Knowledge Bases than the biological facts.

6. The rest of the 186 questions are either of the types mentioned in section 2.4 or complex questions (less than 25 questions), which can be broken down to simpler questions of the aforementioned types. For example, "How have prokaryotic cells given origin to aerobic eukaryotic cells and to photosynthetic aerobic eukaryotic cells?"

While we have a good coverage on "How"questions in the collection of 1,800 questions, there are still some challenges that need to be addressed.

1. The relation R between X and Y of the question "How does X R Y?" could be complex. While determining R based on graph structures like KDG is easier than on unstructured text, it is still a challenge for future work.

2. The collection of question is by no means exhausted. New types of "How"questions obviously will come, and we will face similar challenges to the case of "Why"questions, discussed below.

As we discussed in section 2.7.3, Verberne (Verberne, 2006) listed four subtypes (possibly overlapped) of "Why"questions asking about *reason*: cause, motivation, circumstance, and purpose. The problem is that this method of categorizing questions into those subtypes and content of the answers are not discussed. Moreover, there could be many more types of "Why"questions. To properly solve the problem, we think much more pioneer research (from many fields such as computer science and philosophy) are needed to:

1. List and categorize types of deep questions.

2. Identify the components and properties of the answers to those question types. This needs to be done from both a social science/philosophy and computer science perspective like we observed in the case of "How does X work?"

From there, people can create the corpus of deep questions and their answers, so that the machine learning techniques can be used to automate the question answering process. Once we have "Why"questions categorized and their answers defined, we can analyze to see if our formulations can be used to represent the answer.

7.3.2 Working with AURA Knowledge Base

The AURA Knowledge Base, in fact, does not satisfy the DAG assumption we need for KDGs. Hence we have to do several preprocessing techniques.

We found thousands of cases in AURA where there two or more different instances are cloned from each other; in other words, there is a circle in clone-from relation. For example, instance b is cloned from a, c is cloned from b, and a is cloned from c. We consider that all such instances are equivalent. Hence we choose one instance (i.e. a) to replace all appearances of other instances (of b or c).

In AURA, a specific relation between two instances is always defined as a pair of predicates R/R'. i.e. ($A\ has-part\ B$) and ($B\ is-part-of\ A$) are two relations from A to B

and from B to A; R and R' are $has-part$ and $is-part-of$. While creating a rooted KDG from AURA, we can make it a DAG by some steps:

1. Start from the root (given input), its order is 0.
2. Extend other nodes, assign them an increasing order such as 1,2,3, etc.
3. Remove all edges that "violate" the topological order. i.e. from a node with higher order to a node with lower order.

Please note that KDGs only acyclic with respect to cpaths (consist of compositional edges, participant edges, locational edges and class edges) can have cycles with respect to behavioral edges. So if we have a cyclic series of events, $E1$ and $E2$, where $E1$ causes $E2$ and $E2$ cause $E1$, we can naturally represented by a loop of behavioral edges from $E1$ to $E2$ and from $E2$ back to $E1$.

7.3.3 From Natural Language to KDGs

Automatically constructing KDGs (or the knowledge bases used to build KDGs), presumably from text, is a tough challenge that required years to be conquered as shown in its first step, representing natural language sentences. As Chaudhri stated in (Chaudhri *et al.*, 2011), representing natural language sentences need more knowledge than what the sentence explicitly contains. We illustrate this idea with the iceberg analogy in Figure 7.2. The tip of the iceberg is Vocabulary Knowledge and Explicitly Stated Knowledge, which is respectively the words in the sentence and the sentence's logical restatement. Hidden underwater is the Linguistic and Background Knowledge needed to really understand the sentence.

Systems built by NL2KR will operate in the first two layers of the iceberg, above water; they can only translate explicit knowledge represented by words in the text. They are not sufficient for the task translating from text to KDGs since this task requires at least all four

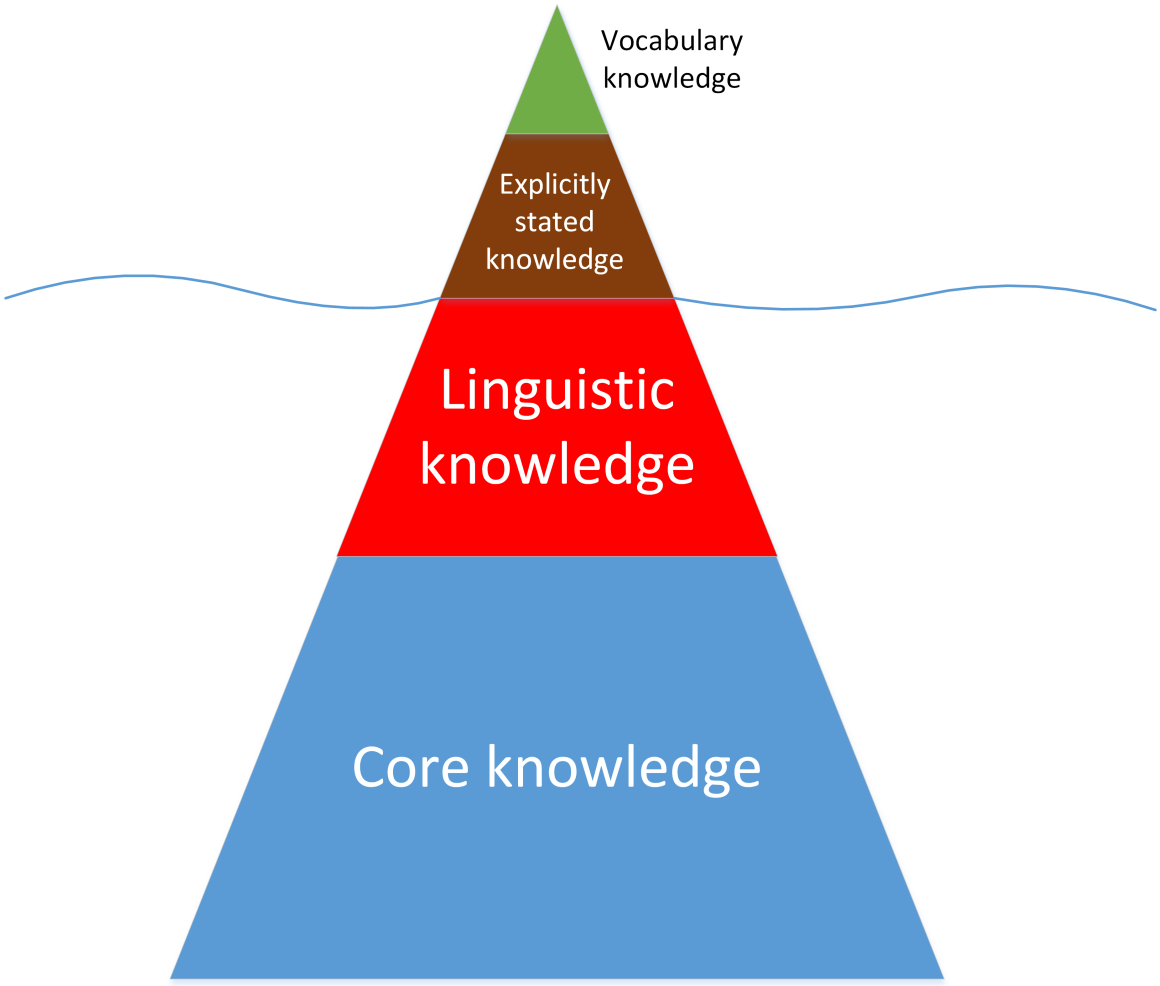

Figure 7.2: Iceberg of knowledge needed to represent a natural language sentence.

Vocabulary Knowledge: class, relation and function symbols representing each word (phrase) in the sentence.

Explicitly Stated Knowledge: logical representation of the sentence using the vocabulary knowledge.

Linguistic Knowledge: English knowledge required to understand the sentence.

Core Knowledge (Background Knowledge): all other knowledge needed to understand the sentence.

layers of the iceberg. Besides solving challenges from language sides such as pronoun resolution or determining the category and hierarchy of instance mentioned in the text, it also need to overcome many other challenges related to knowledge representation - such as figuring out the unrelated data to filter out, determining the default and exception cases. For example, take the given excerpt below. It needs to determine (1) which class/instance should have the encoding of Calvin cycle: plant, algae, cyanobacteria, or some others; (2) how to represent Krebs cycle for ***some*** bacteria, but not all of them.

"In plants, algae and cyanobacteria, sugars are produced by a subsequent sequence of light-independent reactions called the Calvin cycle, but some bacteria use different mechanisms, such as the reverse Krebs cycle."

The K-Parser (Sharma *et al.*, 2015b,c,a) had been developed as our first attempt to address some of the aforementioned challenges. Its ultimate goal is to parse natural text, obtain some knowledge of the bottom two layers of the iceberg, presenting them in a format like KDGs. However, K-Parser has obviously not yet entirely solved the task of translating natural language to KDGs, an extremely challenging task that we believe will be active for a long time.

7.3.4 Other Types of Evaluation

In chapter 2, we presented our three ways of evaluating the formulation of deep questions: by examples, conceptually by their components, and by their properties. Here, we discuss about another possible way to evaluate: creating a corpus of deep questions and their answers, and comparing the answers given by the "system" and gold standard answer, automatically or manually by a human reader. There are still a lot of challenges to be solved before we can have the whole end-to-end system to answer deep questions; but when we have a whole system, we can evaluate it by the same method.

According to the best of our knowledge, there is no research categorizing deep ques-

tions such as “How” and “Why” with respect to the answers they demand, and no corpus has been created as such for evaluating such results. Hence the first challenge of this evaluation approach is to create a corpus of deep questions and their answers. This first task itself has several challenges.

Since the answers later must come from a knowledge base, we must use some knowledge bases that possibly contain the answers to deep questions. Unfortunately, almost all of the knowledge bases we investigated are not built for answering deep questions and do not have the necessary information to answer them (i.e. events, sub-events, event-event relations, etc.) The exceptions are the knowledge base we are using (AURA, biology field) and similar ones (i.e AURA in chemistry and physics), but they have very limited coverage. Coming up with the questions now requires effort to manually analyze the knowledge base to find the intersection of the questions we can possibly have; the formulation to answer and the questions that the KB contains for the answers.

Moreover, in deciding if the knowledge base may contain the answers, there is one more crucial prerequisite: what criteria of “correct” answers should be adopted. Identifying such “correct” answers is exactly what we were trying to do in chapter 2.

While we do not have enough resource to overcome all of those challenges, we decided that we should postpone this type of evaluation, at least until our work here - which paves a way for formalizing answers to deep questions - is publicly accepted.

In chapter 4 and the published paper that it came from, our contribution and focus are proposing the ASP rules in deriving missing information. We, hence, evaluated our results by formal proves. In future work when we are working on a whole end-to-end system with a bigger knowledge base, we could use another evaluation method. We could build a gold standard corpus of information that should be enhanced, then compare our results of the gold standard to have precision and recall. However, those results are not always comparable to the ones of related works (please see chapter 4 for more details), as the

technique and performance are tied to the KB's representation. Considering the efforts and the benefits of the "machine learning" style evaluation at this point, we held it off and focus on emergent topics.

7.3.5 Future Work

In chapter 2, we defined the answers to several deep questions. For future work, we will expand to many more types of deep questions. First, we plan to continue working on defining the answers to some "Why" questions mentioned in section 2.7.3. Those questions asked for cause ("Why did the flower get dry?"), motivation "Why do you water the flower?"), circumstance ("Why should we be able to finish this today?") and purpose ("Why do people have eyebrows?"). Secondly, we are going to automatically identify the label of a path in a KDG. For example, based on the semantics of its nodes and edges, we automatically identify if a path is explaining "create" or "help" properties.

Another possible future project is constructing a larger KDG database, possibly from text. From there, we can further pursue aforementioned Question Answering approaches or work on new NLU applications such as reading comprehension or the Winograd challenge.

For the NL2KR system, we plan to improve the Inverse Lambda theory to cover more cases, further automatize the creation of an initial dictionary, and use word sense disambiguation to improve translation.

7.4 Summary

- In Chapter 1, we introduced the motivation and listed specific contributions of the dissertation.

- In Chapter 2, we defined the KDG structure and its sub-structures to answer some deep questions. We also investigated the properties of those structures.

- In Chapter 3, we proposed formulations and algorithms to solve the cloning and uni-

fication problem. Additionally, we reimplemented and improved the factual Question Answering in (Baral and Liang, 2012) using our new formulations.

- In Chapter 4, we showed our implementations of KDG and deep Question Answering using Answer Set Programming.

- In Chapter 5, we talked about the reasoning needed to complete the missing information in knowledge bases used to construct KDGs.

- In Chapter 6, we introduced the NL2KR system, which can be used to build systems that translate natural language to formal language.

- In Chapter 7, we concluded the contributions in this dissertation, and we discussed the applications to Natural Language Understanding and possible directions for future work.

REFERENCES

Altman, R., M. Buda, X. Chai, M. Carillo, R. Chen and N. Abernethy, "RiboWeb: An ontology-based system for collaborative molecular biology", Intelligent Systems and Their Applications, IEEE **14**, 5, 68–76 (1999).

Aouladomar, F., "Towards answering procedural questions", in "Proceedings of the IJCAI Workshop on Knowledge and Reasoning for Answering Questions", pp. 21–31 (2005).

Aron, J., "How innovative is Apple's new voice assistant, Siri?", The New Scientist **212**, 2836, 24, URL `http://www.sciencedirect.com/science/article/pii/S026240791162647X` (2011).

Artzi, Y., N. FitzGerald and L. S. Zettlemoyer, "Semantic parsing with combinatory categorial grammars.", in "ACL (Tutorial Abstracts)", p. 2 (2013).

Artzi, Y. and L. Zettlemoyer, "UW SPF: The University of Washington Semantic Parsing Framework", arXiv preprint arXiv:1311.3011 (2013a).

Artzi, Y. and L. Zettlemoyer, "Weakly supervised learning of semantic parsers for mapping instructions to actions.", TACL **1**, 49–62 (2013b).

Bancilhon, F., D. Maier, Y. Sagiv and J. Ullman, "Magic sets and other strange ways to implement logic programs", in "Proceedings of the fifth ACM SIGACT-SIGMOD symposium on Principles of database systems", pp. 1–15 (ACM, 1985).

Baral, C., *Knowledge representation, reasoning and declarative problem solving* (Cambridge university press, 2003).

Baral, C. and J. Dzifcak, "Solving Puzzles Described in English by Automated Translation to Answer Set Programming and Learning How to Do that Translation", in "KR", (2012).

Baral, C., J. Dzifcak, M. A. Gonzalez and J. Zhou, "Using Inverse lambda and Generalization to Translate English to Formal Languages", CoRR **abs/1108.3843** (2011).

Baral, C., J. Dzifcak and T. C. Son, "Using Answer Set Programming and Lambda Calculus to Characterize Natural Language Sentences with Normatives and Exceptions.", in "AAAI", edited by D. Fox and C. P. Gomes, pp. 818–823 (AAAI Press, 2008), URL `http://dblp.uni-trier.de/db/conf/aaai/aaai2008.html#BaralDS08`.

Baral, C., M. A. Gonzalez and A. Gottesman, "Correct Reasoning", in "Correct Reasoning", edited by E. Erdem, J. Lee, Y. Lierler and D. Pearce, chap. The Inverse Lambda Calculus Algorithm for Typed First Order Logic Lambda Calculus and Its Application to Translating English to FOL, pp. 40–56 (Springer-Verlag, Berlin, Heidelberg, 2012a), URL `http://dl.acm.org/citation.cfm?id=2363344.2363348`.

Baral, C. and S. Liang, "From Knowledge Represented in Frame-based Languages to Declarative Representation and Reasoning via ASP", 13th International Conference on Principles of Knowledge Representation and Reasoning (2012).

Baral, C. and N. H. Vo, "Event-Object Reasoning with Curated Knowledge Bases: Deriving Missing Information", arXiv preprint arXiv:1306.4411 (2013).

Baral, C., N. H. Vo and S. Liang, "Answering Why and How questions with respect to a frame-based knowledge base: a preliminary report", Technical Communications of the 28th International Conference on Logic Programming (ICLP'12) **17**, 26–36, URL http://drops.dagstuhl.de/opus/volltexte/2012/3607 (2012b).

Blackburn, P. and J. Bos, *Representation and Inference for Natural Language: A First Course in Computational Semantics* (Center for the Study of Language and Information, Stanford, CA, 2005).

Bobrow, D. G., "Natural language input for a computer problem solving system", (1964).

Bollacker, K., C. Evans, P. Paritosh, T. Sturge and J. Taylor, "Freebase: a collaboratively created graph database for structuring human knowledge", in "Proceedings of the 2008 ACM SIGMOD international conference on Management of data", pp. 1247–1250 (2008), URL http://dl.acm.org/citation.cfm?id=1376746.

Brewka, G., "The logic of inheritance in frame systems", in "Intl. Joint Conference on Artificial Intelligence (IJCAI)", pp. 483–488 (1987).

Brizan, D. G. and A. U. Tansel, "A survey of entity resolution and record linkage methodologies", Communications of the IIMA **6**, 3, 41–50 (2006).

Bunge, M., "How does it work? The search for explanatory mechanisms", Philosophy of the social sciences **34**, 2, 182–210, URL http://pos.sagepub.com/content/34/2/182.short (2004).

Chaudhri, V., A. Goldenkranz, R. Fikes and P. Seyed, "What is hard about representing biology textbook knowledge", in "Proceedings of the sixth international conference on Knowledge capture", pp. 185–186 (2011).

Chaudhri, V. K., B. Cheng, A. Overtholtzer, J. Roschelle, A. Spaulding, P. Clark, M. Greaves and D. Gunning, "Inquire biology: A textbook that answers questions.", AI Magazine **34**, 3, 55–72, URL http://www.questia.com/magazine/1G1-346925127/inquire-biology-a-textbook-that-answers-questions (2013).

Chaudhri, V. K., P. E. Clark, S. Mishra, J. Pacheco, A. Spaulding and J. Tien, "Aura: Capturing knowledge and answering questions on science textbooks", Tech. rep., Technical report, SRI International (2009).

Chaudhri, V. K., R. E. Fikes and Z. M. Zadeh, "Analysis of How and Why Questions about a Biology Textbook", in "Proceedings of the 2nd Deep Knowledge Representation and Reasoning Challenge Workshop. Playa Vista, CA", (2012).

Chaudhri, V. K. and T. C. Son, "Specifying and Reasoning with Underspecified Knowledge Bases Using Answer Set Programming.", in "KR", (2012), URL http://www.aaai.org/ocs/index.php/KR/KR12/paper/viewPDFInterstitial/4524/4912.

Chen, D. L. and R. J. Mooney, "Learning to interpret natural language navigation instructions from observations.", in "AAAI", vol. 2, pp. 1–2 (2011).

Chrisley, R. and S. Begeer, *Artificial intelligence: critical concepts*, vol. 1 (Taylor & Francis, 2000).

Church, A., "An Unsolvable Problem of Elementary Number Theory", American Journal of Mathematics **58**, 2, 345–363, URL `http://dx.doi.org/10.2307/2371045` (1936).

Clark, P. and B. Porter, *KM (v2.0 and later): Users Manual* (2011).

Clark, P., B. Porter and B. Works, "KM: The knowledge machine 2.0: Users manual", (2004).

Clark, S. and J. R. Curran, "Wide-coverage Efficient Statistical Parsing with CCG and Log-linear Models", Comput. Linguist. **33**, 4, 493–552, URL `http://dx.doi.org/10.1162/coli.2007.33.4.493` (2007).

Costantini, S. and A. Paolucci, "Towards Translating Natural Language Sentences into ASP.", in "CILC", (2010).

Curran, J., S. Clark and J. Bos, "Linguistically Motivated Large-Scale NLP with C&C and Boxer", in "Proceedings of the 45th Annual Meeting of the Association for Computational Linguistics Companion Volume Proceedings of the Demo and Poster Sessions", pp. 33–36 (Association for Computational Linguistics, Prague, Czech Republic, 2007), URL `http://www.aclweb.org/anthology/P07-2009`.

DKRC2011, "Deep Knowledge Representation Challenge", https://sites.google.com/site/dkrckcap2011 (2011).

DKRC2012, "2nd Deep Knowledge Representation Challenge", URL `https://sites.google.com/site/2nddeepkrchallenge` (2012).

Dzifcak, J., M. Scheutz, C. Baral and P. Schermerhorn, "What to do and how to do it: Translating natural language directives into temporal and dynamic logic representation for goal management and action execution", in "Robotics and Automation, 2009. ICRA'09. IEEE International Conference on", pp. 4163–4168 (IEEE, 2009).

Fan, W., F. Geerts and X. Jia, "Conditional dependencies: A principled approach to improving data quality", in "Dataspace: The Final Frontier", pp. 8–20 (Springer, 2009).

Ferrucci, D., E. Brown, J. Chu-Carroll, J. Fan, D. Gondek, A. Kalyanpur, A. Lally, J. Murdock, E. Nyberg, J. Prager *et al.*, "Building Watson: An overview of the DeepQA project", AI Magazine **31**, 3, 59–79 (2010).

Fikes, R. and T. Kehler, "The role of frame-based representation in reasoning", Communications of the ACM **28**, 9, 904–920 (1985).

Friedland, N. S., P. G. Allen, G. Matthews, M. Witbrock, D. Baxter, J. Curtis, B. Shepard, P. Miraglia, J. Angele, S. Staab, E. Moench, H. Oppermann, D. Wenke, D. Israel, V. Chaudhri, B. Porter, K. Barker, J. Fan, S. Y. Chaw, P. Yeh, D. Tecuci and P. Clark, "Project Halo: Towards a Digital Aristotle", AI Magazine **25**, 4, 29–48, URL `http://www.projecthalo.com/content/docs/aim_final_submission.pdf` (2004).

Gatt, A. and E. Reiter, "Simplenlg: A realisation engine for practical applications", in "Proceedings of the 12th European Workshop on Natural Language Generation", pp. 90–93 (Association for Computational Linguistics, 2009).

Ge, R. and R. J. Mooney, "A statistical semantic parser that integrates syntax and semantics", in "Proceedings of the Ninth Conference on Computational Natural Language Learning", pp. 9–16 (Association for Computational Linguistics, 2005).

Ge, R. and R. J. Mooney, "Learning a compositional semantic parser using an existing syntactic parser", in "Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2", pp. 611–619 (Association for Computational Linguistics, 2009).

Gebser, M., R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub and S. Thiele, "A user's guide to gringo, clasp, clingo, and iclingo", November **77**, 78–80 (2008).

Gelfond, M. and V. Lifschitz, "The stable model semantics for logic programming", in "Logic Programming: Proc. of the Fifth Int'l Conf. and Symp.", edited by R. Kowalski and K. Bowen, pp. 1070–1080 (MIT Press, 1988).

Gerring, J., "Causal mechanisms: Yes, but", Comparative Political Studies **43**, 11, 1499–1526 (2010).

Getoor, L. and C. P. Diehl, "Link mining: a survey", ACM SIGKDD Explorations Newsletter **7**, 2, 3–12 (2005).

Gonzalez, M. A., *An Inverse Lambda Calculus Algorithm*, Ph.D. thesis, Arizona State University (2010).

Google, "www.google.com/landing/now/", (2012).

Gunning, D., V. K. Chaudhri, P. E. Clark, K. Barker, S.-Y. Chaw, M. Greaves, B. Grosof, A. Leung, D. D. McDonald, S. Mishra *et al.*, "Project Halo Update - Progress Toward Digital Aristotle", AI Magazine **31**, 3, 33–58 (2010).

Gupta, P., V. Gupta, M. T. Yousef, E. M. Saad, S. M. Habashy, A. Jegatheesan, J. Murugan, B. Neelagantaprasad, G. Rajarajan and S. Goel, "A Survey of Text Question Answering Techniques", International Journal of Computer Applications **53**, 4, 1–8 (2012).

Herzog, T. N., F. J. Scheuren and W. E. Winkler, *Data quality and record linkage techniques* (Springer, 2007).

Higashinaka, R. and H. Isozaki, "Corpus-based question answering for why-questions", Proc. of IJCNLP **1**, 418–425 (2008).

Hockenmaier, J. and M. Steedman, "CCGbank: A Corpus of CCG Derivations and Dependency Structures Extracted from the Penn Treebank", Comput. Linguist. **33**, 3, 355–396, URL http://dx.doi.org/10.1162/coli.2007.33.3.355 (2007).

Hoffmann, J. *et al.*, "The Metric-FF Planning System: Translating "Ignoring Delete Lists" to Numeric State Variables", J. Artif. Intell. Res.(JAIR) **20**, 291–341 (2003).

Horty, J., "Some direct theories of non-monotonic inheritance", in "Handbook of Logic in AI and logic programming", edited by D. Gabbay and C. Hogger (1994).

Karp, P., M. Riley, M. Saier, I. Paulsen, J. Collado-Vides, S. Paley, A. Pellegrini-Toole, C. Bonavides and S. Gama-Castro, "The ecocyc database", Nucleic acids research **30**, 1, 56 (2002).

Kate, R. J. and R. J. Mooney, "Using string-kernels for learning semantic parsers", in "Proceedings of the 21st International Conference on Computational Linguistics and the 44th annual meeting of the Association for Computational Linguistics", pp. 913–920 (Association for Computational Linguistics, 2006).

Kifer, M. and G. Lausen, "F-Logic: A Higher-Order language for Reasoning about Objects, Inheritance, and Scheme", in "SIGMOD Conference", pp. 134–146 (1989).

Krishnamurthy, J. and T. Kollar, "Jointly learning to parse and perceive: Connecting natural language to the physical world.", TACL **1**, 193–206 (2013).

Krishnamurthy, J. and T. M. Mitchell, "Weakly supervised training of semantic parsers", in "Proceedings of the 2012 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning", pp. 754–765 (Association for Computational Linguistics, 2012), URL http://dl.acm.org/citation.cfm?id=2391030.

Kumbhare, K., "Robust implementation of NL2KR System and its application in iRODS domain", (2013).

Kushman, N. and R. Barzilay, "Using semantic unification to generate regular expressions from natural language", in "North American Chapter of the Association for Computational Linguistics (NAACL)", (North American Chapter of the Association for Computational Linguistics (NAACL), 2013), URL http://dspace.mit.edu/handle/1721.1/79645.

Kwiatkowski, T., E. Choi, Y. Artzi and L. Zettlemoyer, "Scaling semantic parsers with on-the-fly ontology matching", homes.cs.washington.edu (2013).

Kwiatkowski, T., L. Zettlemoyer, S. Goldwater and M. Steedman, "Inducing probabilistic CCG grammars from logical form with higher-order unification", in "Proceedings of the 2010 conference on empirical methods in natural language processing", pp. 1223–1233 (Association for Computational Linguistics, 2010).

Kwiatkowski, T., L. Zettlemoyer, S. Goldwater and M. Steedman, "Lexical generalization in ccg grammar induction for semantic parsing", in "Proceedings of the Conference on Empirical Methods in Natural Language Processing", pp. 1512–1523 (Association for Computational Linguistics, 2011), URL http://dl.acm.org/citation.cfm?id=2145593.

Leone, N., G. Pfeifer, W. Faber, T. Eiter, G. Gottlob, S. Perri and F. Scarcello, "The DLV system for knowledge representation and reasoning", ACM Transactions on Computational Logic (TOCL) **7**, 3, 499–562 (2006).

Levesque, H. J., E. Davis and L. Morgenstern, "The winograd schema challenge.", in "KR", (2012).

Liang, P., M. I. Jordan and D. Klein, "Learning dependency-based compositional semantics", Computational Linguistics **39**, 2, 389–446, URL http://www.mitpressjournals.org/doi/abs/10.1162/COLI_a_00127 (2013).

Lierler, Y. and P. Schüller, "Parsing combinatory categorial grammar via planning in answer set programming", in "Correct Reasoning", pp. 436–453 (Springer, 2012).

Liu, H. and P. Singh, "Conceptneta practical commonsense reasoning tool-kit", BT technology journal **22**, 4, 211–226, URL http://link.springer.com/article/10.1023/B:BTTJ.0000047600.45421.6d (2004).

Matuszek, C., J. Cabral, M. J. Witbrock and J. DeOliveira, "An introduction to the syntax and content of cyc.", in "AAAI Spring Symposium: Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering", pp. 44–49 (Citeseer, 2006).

Matuszek, C., E. Herbst, L. Zettlemoyer and D. Fox, "Learning to parse natural language commands to a robot control system", in "Experimental Robotics", pp. 403–415 (Springer, 2013), URL http://link.springer.com/chapter/10.1007/978-3-319-00065-7_28.

Maybury, M., *New directions in question answering* (AAAI press, 2004).

McCorduck, P., "Machines who think", (2004).

McDermott, D., M. Ghallab, A. Howe, C. Knoblock, A. Ram, M. Veloso, D. Weld and D. Wilkins, "PDDL-the planning domain definition language", Tech. rep. (1998).

Mendes, A. C. and L. Coheur, "When the answer comes into question in question-answering: survey and open issues", Natural Language Engineering **1**, 1, 1–32, URL http://journals.cambridge.org/abstract_S1351324911000350 (2012).

Miller, G. A., "WordNet: a lexical database for English", Communications of the ACM **38**, 11, 39–41 (1995).

Minsky, M., "A Framework for Representing Knowledge", in "The Psychology of Computer Vision", edited by P. Winston, pp. 211–277 (McGraw-Hill, New York, 1975).

Moldovan, D., S. Harabagiu, M. Pasca, R. Mihalcea, R. Girju, R. Goodrum and V. Rus, "The structure and performance of an open-domain question answering system", in "Proceedings of the 38th Annual Meeting on Association for Computational Linguistics", pp. 563–570 (Association for Computational Linguistics, 2000), URL `http://dl.acm.org/citation.cfm?id=1075289`.

Montague, R., "English as a Formal Language", in "Formal Philosophy: Selected Papers of Richard Montague", edited by R. H. Thomason, pp. 188–222 (Yale University Press, New Haven, London, 1974).

Moss, L. and D. J. Nicholson, "On nature and normativity: normativity, teleology, and mechanism in biological explanation", Studies in History and Philosophy of Science Part C: Studies in History and Philosophy of Biological and Biomedical Sciences **43**, 1, 88–91 (2012).

Niemelä, I. and P. Simons, "Smodels—an implementation of the stable model and well-founded semantics for normal logic programs", Logic Programming and Nonmonotonic Reasoning pp. 420–429 (1997).

Quirk, R., D. Crystal and P. Education, *A comprehensive grammar of the English language*, vol. 397 (Cambridge Univ Press, 1985).

Rahm, E. and H. H. Do, "Data cleaning: Problems and current approaches", IEEE Data Engineering Bulletin **23**, 4, 3–13 (2000).

Russell, S. J., P. Norvig, J. F. Canny, J. M. Malik and D. D. Edwards, *Artificial intelligence: a modern approach*, vol. 2 (Prentice hall Englewood Cliffs, 1995).

Sacerdoti, E., "Planning in a hierarchy of abstraction spaces", Artificial Intelligence **5**, 115–135 (1974).

Schank, R. C. and L. Tesler, "A conceptual dependency parser for natural language", in "Proceedings of the 1969 conference on Computational linguistics", pp. 1–3 (Association for Computational Linguistics, 1969).

Sedgewick, R. and K. Wayne, *Algorithms, 4th Edition.* (Addison-Wesley, 2011).

Sharma, A., N. H. Vo, S. Aditya and C. Baral, "Identifying various kinds of event mentions in k-parser output", The 3rd Workshop on EVENTS: Definition, Detection, Coreference, and Representation. HLT-NAACL (2015a).

Sharma, A., N. H. Vo, S. Aditya and C. Baral, "Towards addressing the winograd schema challenge-building and using a semantic parser and a knowledge hunting module", IJCAI (2015b).

Sharma, A., N. H. Vo, S. Gaur and C. Baral, "An approach to solve winograd schema challenge using automatically extracted commonsense knowledge", in "2015 AAAI Spring Symposium Series", (2015c).

Socher, R., J. Bauer, C. D. Manning and A. Y. Ng, "Parsing with Compositional Vector Grammars", in "ACL (1)", pp. 455–465 (2013).

Speer, R. and C. Havasi, "Conceptnet 5: A large semantic network for relational knowledge", in "The Peoples Web Meets NLP", pp. 161–176 (Springer, 2013), URL http://link.springer.com/chapter/10.1007/978-3-642-35085-6_6.

Steedman, M., *The Syntactic Process* (MIT Press, Cambridge, MA, 2000).

Strzalkowski, T. and S. Harabagiu, *Advances in open domain question answering* (Springer Dordrecht,, Netherlands, 2006).

Tang, L. R. and R. J. Mooney, "Using multiple clause constructors in inductive logic programming for semantic parsing", in "Machine Learning: ECML 2001", pp. 466–477 (Springer, 2001).

Touretzky, D., *The mathematics of inheritance systems* (Morgan Koufmann, Los Altos, Ca., 1986).

Toutanova, K., D. Klein, C. D. Manning and Y. Singer, "Feature-rich part-of-speech tagging with a cyclic dependency network", in "Proceedings of the 2003 Conference of the North American Chapter of the Association for Computational Linguistics on Human Language Technology - Volume 1", (2003).

Verberne, S., "Developing an approach for *why*-question answering", in "Proceedings of the Eleventh Conference of the European Chapter of the Association for Computational Linguistics: Student Research Workshop", EACL '06, pp. 39–46 (Association for Computational Linguistics, Stroudsburg, PA, USA, 2006), URL http://dl.acm.org/citation.cfm?id=1609039.1609044.

Verberne, S., *In Search of the Why*, Ph.D. thesis, Ph. D. Dissertation, University of Nijmegen, Oxford, UK (2009).

Weizenbaum, J., "Computer power and human reason: From judgment to calculation.", (1976).

Wong, Y. W. and R. J. Mooney, "Learning for semantic parsing with statistical machine translation", in "Proceedings of the main conference on Human Language Technology Conference of the North American Chapter of the Association of Computational Linguistics", pp. 439–446 (Association for Computational Linguistics, 2006).

Wong, Y. W. and R. J. Mooney, "Learning synchronous grammars for semantic parsing with lambda calculus", in "Annual Meeting-Association for computational Linguistics", vol. 45, p. 960 (Citeseer, 2007).

Woods, W. A., "Transition network grammars for natural language analysis", Communications of the ACM **13**, 10, 591–606 (1970).

Wouters, A., "The functional perspective of organismal biology", Current Themes in Theoretical Biology pp. 33–69 (2005).

Zelle, J. M. and R. J. Mooney, "Learning to Parse Database Queries Using Inductive Logic Programming", in "Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 2", AAAI'96, pp. 1050–1055 (AAAI Press, 1996), URL http://dl.acm.org/citation.cfm?id=1864519.1864543.

Zettlemoyer, L. S. and M. Collins, "Learning to Map Sentences to Logical Form: Structured Classification with Probabilistic Categorial Grammars", in "UAI", pp. 658–666 (AUAI Press, 2005), URL http://dblp.uni-trier.de/db/conf/uai/uai2005.html#ZettlemoyerC05.

Zettlemoyer, L. S. and M. Collins, "Online learning of relaxed CCG grammars for parsing to logical form", in "In Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning (EMNLP-CoNLL-2007)", (2007).

Zettlemoyer, L. S. and M. Collins, "Learning context-dependent mappings from sentences to logical form", in "Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 2-Volume 2", pp. 976–984 (Association for Computational Linguistics, 2009).

APPENDIX A

FOR CHAPTER 2

Proof 13 (Proof for proposition 1) *We now prove 2 statements which together prove the proposition.*

1. *$G1$ is a subgraph of $G2$.*
   We are proving all nodes and edges of $G1$ is also in $G2$.
   Let A be a node in $G1$. By definition of rooted KDG, there is a cpath in G from X to A. Since there is also a cpath in G from Z to X, there is a cpath in G from Z to A.

   By definition of rooted KDG, A must be in $KDG(Z)$.

   $\Rightarrow$ $G2$ contains A since $G2$ is constructed from $KDG(Z)$ and there is a cpath from X to A.

   Let E be an edge of $G1$, from A to B. As we have already proved, nodes A and B of $G1$ are also in $KDG(Z)$ and $G2$. By definition of rooted KDGs, the edge E must be in the $KDG(Z)$ and also in $G2$.

   Conclusion: $G1$ is a subgraph of $G2$

2. *$G2$ is a subgraph of $G1$*
   Similary, we are proving all nodes and edges of $G2$ is also in $G1$.

   Let A be a node in $G2$. According to definition of rooted KDG, there is a cpath in $KDG(Z)$ from X to A. Because $KDG(Z)$ is a subgraph of G, that cpath is also in G.

   $\Rightarrow$ There is a cpath in G from X to A.
   $\Rightarrow$ A is a node in $G1$.

   Similar to previous case, any edge E in $G2$ is also in $G1$. Conclusion: $G1$ is a subgraph of $G2$

APPENDIX B

FOR CHAPTER 3

Proof 14 (Lemma 11) *If X is an instance in KB, Π_{KB} does not contain the fact $class(X)$ and at least one fact $has(X,S,V)$ or $has(Y,S,X)$. Rule $a2$, $a3$ give us that $ins(X)$ must be true in all answer set of Π_{KB}.*

If $ins(X)$ is in any answer set of Π_{KB}, it must have been introduced by either rule $a2$ or $a3$. This means that X is not a class in KB: X must be an instance in KB.

Proof 15 (Lemma 12) *Because of the KB's representation, X is an instance-of Y iff Π_{KB} contains $has(X,instance\_of,Y)$ or $has(X,clean\_instance\_of,Y)$. From rules $a4$ and $a5$, it is easy to see that: X is an instance-of Y iff $\Pi_{KB} \models instance\_of(X,Y)$*

Forward direction *Suppose that X is a tc-instance-of Y. According to Definition 20, there exists a set of classes $\{C_1,C_2,...,C_k = Y\}$ where all of the following conditions are satisfied:*

- *x is an instance of C_1*
- *C_i is a subclass of C_{i+1} $(0 < i < k)$*

Since C_i is a subclass of C_{i+1}, the Π_{KB} must contain $has(C_i,superclass,C_{i+1})$.

The rules $a4$ and $a5$ make sure that Π_{KB}'s answer sets contain $instance\_of(X,C_1)$.

The rule $a6$ makes sure that Π_{KB}'s answer sets contain $tc\_instance\_of(X,C_1)$.

The rule $a7$ makes sure that Π_{KB}'s answer sets contain $tc\_instance\_of(X,C_2)$, and then again contain $tc\_instance\_of(X,C_2)$, then $tc\_instance\_of(X,C_3)$,...,$tc\_instance\_of(X,C_k)$ or $tc\_instance\_of(X,Y)$.

Backward direction *Suppose that an answer set of Π_{KB} contains $tc\_instance\_of(X,Y)$. It must be introduced by either rule $a6$ or $a7$.*

If $tc\_instance\_of(X,Y)$ is introduced by $a6$, the answer set must also contain $instance\_of(X,Y)$. So X is a tc-instance-of Y.

If $tc\_instance\_of(X,Y)$ is introduced by $a7$, the answer set must also contain $tc\_instance\_of(X,M_1)$ and $has(M_1,superclass,Y)$. Again, the $tc\_instance\_of(X,M_1)$ must come from either $a6$ or $a7$.

If it comes from $a7$, the answer set must also contain $tc\_instance\_of(X,M_2)$ and $has(M_2,superclass,M_1)$,...

Such classes M_i are limited in the hierarchy graph and let $tc\_instance\_of(X,M_{k-1})$ be the last one of the form that comes from $a7$.

The $tc\_instance\_of(X,M_k)$ in the body of the rule (that gives $tc\_instance\_of(X,M_{k-1})$) must come from $a6$. This means that X is an instance of M_k.

Together with the facts that M_i is the superclass of M_{i-1} $(k \geq i \geq 1)$ and that the answer set contains $has(M_i,superclass,M_{i-1})$, Definition 21 gives us that X is a tc-instance-of Y.

Proof 16 (Lemma 14) *Using Lemma 12, proving this proposition is straight-forward from Definition 21 and rules $a10$,$a11$ and $a12$.*

Proof 17 (Proposition 3) *We will prove that x is a node in $G^*(w)$ iff an answer set of Π_{KB} contains $connect(w,x,w)$. We prove this in two cases:*

- *$w = a$ which is not g-cloned from any node. (Similar to $a1$ in Figure 3.4)*
- *$w = b$ which is g-cloned from a. (Similar to $b1$ in Figure 3.4)*

The case where w = c, c is microcloned from more than one node can be proven similarly to second case above.

In each case, we prove in two directions:

- ***Forward direction:*** *If x is a node in $G^*(w)$, there exists an answer set Π_{KB} that contains $connect(w,x,w)$.*
- ***Backward direction:*** *If an answer set of Π_{KB} contains $connect(w,x,w)$, x is a node in $G^*(w)$.*

Forward direction: *If x is a node in $G^*(a)$. Let ANS is any answer set of Π_{KB}, ANS contains $connect(a,x,a)$.*

1. *When a is not g-cloned from any node, $G^*(a) = G(a)$ (Definition 28).*
2. *Because x is a node in $G(a)$, there exists a path in $G(a)$ from a to x containing nodes $c_1 = a, c_2, ..., c_k = x$. All the edges are inheritable edges, except the last one from c_{k-1} to c_k; it could be either non-inheritable or inheritable.*
3. *ANS contain $has(c_i, S, c_{i+1})$ where ($i = 1..k-1$ and (c_i, S, c_{i+1}) in the edge between c_i and c_{i+1} in $G(a)$ in KB) because that is how KB is represented.*
4. *ANS contain $hasc((c_i, S, c_{i+1}), a)$ because of rule a25.*
5. *If all of the edges in the path are inheritable edges, rule a21 and a22 make ANS contain $connectI(c_0, c_k, a)$. a23 makes ANS contain $connect(c_0, c_k, a)$, which is $connect(a,x,a)$.*
6. *If the last edge in the path is a non-inheritable edge, a21 and a22 make ANS contain $connectI(c_0, c_{k-1}, a)$. a23 makes ANS contain $connect(c_0, c_{k-1}, a)$; then, a24 introduces $connect(c_0, c_k, a)$ into ANS.*

Backward direction: *If the answer set ANS of Π_{KB} contains $connect(a,x,a)$, x is a node in $G^*(a)$.*

1. *When a is not g-cloned from any node, $G^*(a) = G(a)$ (Definition 28).*
2. *$connect(a,x,a)$ in ANS has to be introduced by either a23 or a24. Assuming it is introduced by a24 (the other case is simpler and can be solved similarly), ANS must contain $connectI(a, c_2, a)$, $hasc((c_2, S, x), a)$ and $noninheritable(S)$. $connectI(a, c_2, a)$ must be introduced by a21 and a22.*
3. *If $connectI(a, c_2, a)$ is introduced by a21, $a = c_2$. If $connectI(a, c_2, a)$ is introduced by a22, ANS must contain $connectI(a, c_3, a)$, $hasc((c_3, S_3, c_2), a)$ and not contain $noninheritable(S_3)$. $connectI(a, c_3, a)$ again can be introduced by either a21 or a22,....*
4. *We then have a chain of facts $connectI(a, c_2, a)$, $connectI(a, c_3, a)$, ..., $connectI(a, c_k, a)$ where the last one must be introduced by a21. So, $a = c_k$ must be true. Moreover, ANS contain $hasc((c_i, S_i, c_{i-1}), a)$ ($3 \leq i \leq k$) and do not contain any $noninheritable(S_i)$.*

5. *Thus, there exists a path through inheritable edges from $c_k = a$ to c_2, then through a non-inheritable edge $c_1 = x$. x is a node in $G(a)$.*

Forward direction: *If x is a node in $G^*(b)$. There exists an answer set ANS of Π_{KB} containing $connect(b,x,b)$.*
According to Definition 28, x is introduced into $G^*(b)$ by three cases:

1. *x is in $G(b)$.*
2. *x is the destination node of one edge in $edge^*(v,\{G^*(a)\})$ where v is a node in $G(b)$.*
3. *x is in $G(t)$ of $G^*(a))$ where t is in $edge^*(v,\{G^*(a)\})$.*

The first case, where x is in $G(b)$, can be proven similarly to the case of $G(a)$. We now consider the next two cases. ***Case 2: x is the destination node of one edge in*** *$edge^*(v,\{G^*(a)\})$*
Let (v', S, x) be the edge $G^*(a)$ that contributes (v, S, x) to $edge^*(v,\{G^*(a)\})$; v is microcloned from v'. According to lemma 9, there must exists a path of inheritable edges from C to v.

1. *Let ANS be an answer set of Π_{KB}.*
2. *Let $M = N_{G(b)}(v,S)$ be the set of nodes in $G(b)$ that are connected from v through edges of type S.*
3. *Let $A = \{a\}$ be the set of nodes that b is g-cloned from.*
4. *Let $N = N_{G^*(a)}(A,S) = N_{G^*(a)}(\{a\},S)$, be the set of nodes in $G^*(a)$ which are connected from a through edges of type S.*
5. *x must be in $MIN_{subsume}(N \setminus_{subsume} M)$. This leads to conclusion that there does not exist any node in M that subsumes x.*
6. *Since v in b is microcloned from v' in a, ANS must contain $microclone((v,b),(v',a))$. ANS also must contain $hasc(v',S,x),a)$ because (v', S, x) is an edge in $G^*(a)$. Rule $a16$ makes ANS contain $may\_have((v,S,x),b,(v',a))$.*
7. *Since there does not exist any node in M that subsumes x, ANS does not contain $has(v,S,x')$ and $subsume(x',x)$ ($v' \neq v, x \neq x'$). Rule $a17$ dictates for ANS NOT to contain $may\_not\_have((v,S,x),b)$.*
8. *Since x is the minimal node in $N \setminus_{subsume} M$, there does not exist any other node in the set that subsumes x, but x does not subsume it back. ANS, thus, does not have $may\_not\_have2((v,S,x),b)$, the head of rule $a19$.*
9. *Rule $a20$ makes sure that an answer set ANS of Π_{KB} that contains $edge\_s(v,S,x),b,(v',a))$ exists.*
10. *Rule $a26$ puts $hasc(v,S,x),b)$ into ANS.*
11. *Since a path of inheritable edges from C to v exists, rules $a21$ and $a22$ put $connectI(b,v,b)$, and then $connectI(b,x,b)$ into ANS. Rule $a23$ makes sure ANS contains $connect(b,x,b)$.*

Case 3: x is in $G(t)$ of $G^*(a)$) where t is in $edge^*(v,\{G^*(a)\})$

1. Similarly to previous case, an answer set ANS containing $connectI(b,t,b)$ exists.

2. Because x is in $G(t)$, there exists a path from t to x, where only the last edge can be non-inheritable or inheritable (the others are inheritable).

3. Rule $a22$ puts into ANS the facts $connectI(b,t=t_i,b)$, $connectI(b,t_2,b)$, ..., $connectI(b,t_k,b)$, and t_i is the node on the path from t to x. t_k is the last node before x. Rule $a24$ or the group of $a22$ and $a23$ then puts $connectI(b,x,b)$ into ANS depending on whether the last edge is non-inheritable.

Backward direction: If the answer set ANS of Π_{KB} contains $connect(b,x,b)$, x is a node in $G^*(b)$.
If x is in $G(b)$, then obviously x is in $G^*(b)$. We will consider only the case that x is not in $G(b)$.

1. Similarly to the backward direction proof in case of $G^*(a)$, the fact $connect(b,x,b)$ must be introduced by rules $a23$ or $a24$. Assume that it comes from $a24$ (the other case is simpler and can be solved similarly).

2. Also like the case of $G^*(a)$, rules $a21$, $a22$, $a23$, and $a24$ ensure that there must be a set of nodes $c_1 = x, c_2, c_3, ..., c_k = b$ where ANS satisfies all following conditions:

   (a) Contains $connect(b,c_i,b)$, $1 \leq i \leq k$

   (b) Contains $connectI(b,c_i,b)$, $1 < i \leq k$,

   (c) Contains $hasc((c_2,S_1,c_1),b)$ and $noninheritable(S)$

   (d) Contains $hasc((c_{i+1},S_i,c_i),b)$ but not $noninheritable(S_i)$, $1 < i < k$

3. Since c_1 is not in $G(b)$ but $c_k = b$ is in $G(b)$, c_i ($1 \leq i \leq k$) must come to $G(b)$ due to c_j. Now, let $t = c_{j-1}$ and $v = c_j$. $G(b)$ contains v but not t. ANS contains $hasc((v,S,t),b)$ but not $noninheritable(S)$.

4. Because t and v are in different object graphs $G(a)$ and $G(b)$, Π_{KB} and ANS do not contain $has(v,S,t)$. v is not in $G^*(a)$ and ANS does not contain $hasc((v,S,t),a)$. The fact $hasc((v,S,t),b)$ thus must not come from $a25$ and $a27$, but rather from $a26$. ANS, thus, contains $edge\_s((v,S,t),b,(v',f))$.

5. Rule $a20$ then gives us that ANS contains $may\_have((v,S,t),b,(v',f))$, $not\_mayhave((v,S,t),b)$ and $not\_mayhave2((v,S,t),b)$.

6. $may\_have((v,S,t),b,(v',f))$ is introduced by rule $a16$, which means ANS must contain $hasc((v',S,t),f)$, $micro\_clones((v,b),(v',f))$ and must not contain $noninheritable(S)$.

7. Tracing back $micro\_clones((v,b),(v',f))$ from rules $a13$, $a14$, and $a15$, we conclude that $micro\_clones((b,b),(f,f))$ must be true, which leads to $f = a$ because b is only microcloned from a.

8. Now, we have that v in b is microcloned from v' in a. Because *ANS* does not contain $not\_mayhave((v,S,t),b)$, we can conclude that t is not subsumed by any node t' in $N_{G(b)(v,S)}$. Furthermore, since *ANS* does not contain $not\_mayhave2((v,S,t),b)$, we can conclude that no other node t'' is in $N_{G*(a)(\{v'\},S)} \setminus_{subsume} N_{G(b)}(v,S)$. Hence t is in one $edge\_s^*_b(x,\{G^*(a)\})$ and therefore in $G^*(b)$ (Definition 28).

9. t is the node c_{j-1} in the set $c_1 = x, c_2, ...c_k$. Consider the fact $hasc((c_{j-1},S_{j-2},c_{j-2}),b)$. This fact must come from rule $a25$, $a26$, or $a27$.

   (a) If $hasc((c_{j-1},S_{j-2},c_{j-2}),b)$ comes from $a25$, *KB* contains $has(c_{j-1},S_{j-2},c_{j-2})$ and $hasc((c_{j-1},S_{j-2},c_{j-2}),f)$ will be true for any context f. c_{j-2} is the child of $c_{j-1} = t$ in the $G(t)$ in $G^*(a)$. By Definition 28, c_{j-2} is also in $G^*(b)$.

   (b) If $hasc((c_{j-1},S_{j-2},c_{j-2}),b)$ comes from $a26$, like the case of $c_{j-2} = t$, c_{j-2} is also in $G^*(b)$.

   (c) If $hasc((c_{j-1},S_{j-2},c_{j-2}),b)$ comes from $a27$. *ANS* must contain
   $edge\_s((x,Sx,v),b,(x',f))$,
   $hasc((c_{j-1},S_{j-2},c_{j-2}),f)$,
   $connect(v,c_{j-1},f$,
   $connect(v,c_{j-2},f$. This means that $G(v)$ in $G^*(f)$ is added to $G^*(b)$ because v is in some $edge\_s$ set at x of $G(b)$, and c_{j-2} is a node in $G(v)$ in $G^*(f)$. By Definition 28, c_{j-2} is also in $G^*(b)$.

10. Similarly, we can prove $c_{j-3}, ..., c_1 = x$ is in $G^*(b)$

Proof 18 (Lemma 16) *We are going to prove that x is a node in $G(w)$ iff an answer set of Π_{KB} contains $relate(w,x)$:*

- ***Forward direction:*** *If x is a node in $G(w)$, there exists an answer set Π_{KB} containing $relate(w,x)$.*
- ***Backward direction:*** *If an answer set Π_{KB} contains $relate(w,x)$, x is a node in $G^*(w)$.*

Forward direction: *If x is a node in $G(w)$, let ANS be any answer set of Π_{KB}; ANS contains $relate(w,x)$.*

1. *Because x is a node in $G(w)$, there exists a path in $G(w)$ from w to x containing nodes $c_1 = w, c_2, ..., c_k = x$. All the edges are inheritable, except the last edge from c_{k-1} to c_k, which could be either non-inheritable or inheritable.*
2. *ANS contains $has(c_i,S,c_{i+1})$ where ($i = 1..k-1$ and (c_i,S,c_{i+1}) in the edge between c_i and c_{i+1} in $G(a)$ in KB) because KB is represented as such.*
3. *ANS contains $relatedI(w = c_1, w = c_1)$ because of rule $a29$.*
4. *ANS contains $relatedI(c_1,c_2)$ because of rule $a30$. Similarly, ANS contains $relatedI(c_1,c_3)$,...*
5. *If the edge from c_{k-1} to c_k is inheritable, ANS contains $relatedI(c_1,c_k)$. Otherwise, it only contain up to $relatedI(c_1,c_{k-1})$*

6. *In both cases, rule $a31$ and $a32$ make ANS contain $relate(c_1 = w, c_k = x)$*

Backward direction: *If the answer set ANS of Π_{KB} contains $relate(w,x)$, x is a node in $G(w)$.*

1. *The fact $relate(w,x)$ must be introduced by rules $a31$ or $a32$. Assume that it comes from $a32$ (the other case is simpler and can be solved similarly).*
2. *Rules $a29$, $a30$, $a31$, and $a32$ ensure that there must be a set of nodes $c_1 = x, c_2, c_3, ..., c_k = w$ so ANS satisfies all following conditions:*
   (a) *Contains $relateI(w, c_i, b)$, $1 < i \leq k$*
   (b) *Contains $has(c_2, S_1, c_1), b)$ and $noninheritable(S_1)$*
   (c) *Contains $has(c_{i+1}, S_i, c_i)$ but not $noninheritable(S_i)$, $1 < i < k$*
3. *The path from w to x through the edges $has(c_{i+1}, S_i, c_i)$, $1 \leq i < k$ has:*
   (a) *Last edge as non-inheritable.*
   (b) *All other edges as inheritable.*
4. *By definition, x is in $G(w)$.*

Proof 19 (Proposition 4) *This proof is based on the proof of proposition 3. We will prove that x is a node in $G^*(w)$ iff an answer set of Π'_{KB} contains $connect(w,x,w)$. We prove this in cases of $G(a)$ and $G(b)$, each in two ways:*

- ***Forward direction:*** *If x is a node in $G^*(w)$, there exists an answer set of Π'_{KB} containing $connect(w,x,w)$.*
- ***Backward direction:*** *If an answer set of Π'_{KB} contains $connect(w,x,w)$, x is a node in $G^*(w)$.*

Forward direction: *If x is a node in $G^*(a)$. Let ANS be any answer set of Π'_{KB}; ANS contains $connect(a,x,a)$.*

1. *ANS must contain $context(w)$ (definition 30)*
2. *Rule $a28$ makes sure ANS contain other $context()$. For example, if $w = b$, ANS must also contain $context(a)$, where b is g-cloned from a.*
3. *Each rule $a1$ - $a27$, if modified, would contain a maximum of two facts of the form $context(C)$ and $relate(C,X)$. Given x and $G(w)$, these facts are satisfied in ANS; the modified rules become the original $a1$ - $a27$.*
4. *The proof in this case (Π'_{KB}) is then similar to that of the previous case of Π_{KB}(proving proposition 3)*

Backward direction: *The extra facts of the form $context(C)$ and $relate(C,X)$ only give us more ammunition and do not change the proof of proposition 3.*

B.2 Program Π (Rules - without Modification for Efficiency)

```
noninheritable(prototype_participants;
prototype_participant_of;
prototype_scope;
prototype_of;
cloned_from;
clone_built_from;
instance_of;
clean_instance_of;
big_nodes).

% grab all instances
% class(C) :- has(X, instance_of, C).
ins(X) :- has(X,S,V),
% context(C), relate(C, X),
not class(X).
ins(V) :- has(X,S,V),
% context(C), relate(C, X),
not class(V).

% -------------------------------------------------
% step 0:
% -------------------------------------------------
instance_of(X,Y) :-    has(X,instance_of,Y).
instance_of(X,Y) :-    has(X,clean_instance_of,Y).
tc_instance_of(X,Y) :- instance_of(X,Y).
tc_instance_of(X,Y) :- tc_instance_of(X,M),  has(M, superclass, Y).

% -------------------------------------------------
% step 1: decide - what clones from what
% -------------------------------------------------
g_cloned_from(X,Y) :- X!=Y, has(X,cloned_from,Y).
g_cloned_from(X,Y) :- X!=Y, has(X,clone_built_from,Y).
g_cloned_from(X,Y) :- X!=Y, tc_instance_of(X,M), has(Y,prototype_of,
   M).


% -------------------------------------------------
% step 2: Sumsume. V1 subsumes V2 if V1 is more specific.
% -------------------------------------------------
% V1 does not subsume V2 if there exist a class C that V2 is tc-
   instance-of but V1 is not.
not_subsume(V1,V2) :-
ins(V1), ins(V2),
tc_instance_of(V2,C),
not tc_instance_of(V1,C).

subsume(V1,V2) :-
ins(V1), ins(V2),
not not_subsume(V1,V2).

% -------------------------------------------------
% step : Define microclones
% -------------------------------------
% We let V1 microclones from V2 if V1 subsumes it, so later V1 gets
   expanded.
```

```
% microclones((X1,C), (X2,F)) reads as X1 in C is microcloned from
   X2 in F.
% Case 1:
microclones((X2,X2), (X1,X1)) :-
% context(X1), context(X2),
ins(X1), ins(X2),
g_cloned_from(X2,X1).

% Case 2:
microclones((V2,C), (V1,F)) :-
% context(C), context(F),
ins(X1), ins(X2),
ins(V1), ins(V2),
microclones((X2,C), (X1,F)),
has(X1, S, V1),
has(X2, S, V2),
not noninheritable(S),
subsume(V1,V2).

% Case 3:
microclones((V2,C), (V1,F1)) :-
% context(C), context(F1), context(F2),
ins(X1), ins(X2),
ins(V1), ins(V2),
microclones((X,C), (X1,F1)),
microclones((X,C), (X2,F2)),
has(X1, S, V1),
has(X2, S, V2),
X1 != X2,
not noninheritable(S),
subsume(V1,V2).


% % Define edges^*_C(x)
% % -------------------------------------
% % may_have(X, S, V, C) reads as (X, S, V) is in edges^*_C(x).

% X may have value V from Y if: X microclones value V from some
   instance Y
% may_have((X,S,V),C,(Y,F)): we may have (X,S,V) in C from Y in F
may_have((X,S,V),C,(Y,F)) :-
% context(C), context(F),
hasc((Y,S,V),F),
microclones((X,C), (Y,F)),
not noninheritable(S).

% Result of set difference with respect to subsume: KB has the edge
   (X, S, V1), may have (X, S, V2) from Xj but V1 subsume V2.
% V is in N \_{subsume} M if may_have ((X,S,V), C,(Y,F)) and not
   may_not_have((X, S, V), C)
may_not_have((X, S, V2), C) :-
% context(C), context(F),
has(X, S, V1), may_have((X, S, V2), C, (Xj, F)),
subsume(V1, V2),
X != Xj,
V1 != V2.
```

```
% Select the most specific node(s)
% -------------------------------------
% Equally specific nodes
equ((X, S, V1), (Xi, Fi), (X, S, V2), (Xj, Fj), C) :-
may_have((X, S, V1), C, (Xi, Fi)), may_have((X, S, V2), C, (Xj, Fj))
   ,
subsume(V2, V1), subsume(V1, V2),
not may_not_have((X, S, V1), C), not may_not_have((X, S, V2), C),
Xi != X, Xj != X, Xi != Xj.

% Most specific nodes: nodes that are not subsumed by any node,
   except its equally specific ones.
may_not_have2((X, S, V1), C) :-
may_have((X, S, V1), C, (Xi,_x)), may_have((X, S, V2), C, (Xj, _)),
subsume(V2, V1), not_subsume(V1, V2),
V1 != V2, Xi != X, Xj != X.

% Because of the imperfection of AURA KB, we have to remove the
   upperbound of the rule to make it work in some cases.
1 { edges_s((X, S, V1), C, (Xj, Fj)): equ((X, S, V), (Xi, Fi), (X, S
   , V1), (Xj, Fj), C);
edges_s((X, S, V), C, (Xi, Fi))} 2 :-
may_have((X, S, V), C, (Xi, Fi)),
not may_not_have((X, S, V), C),
not may_not_have2((X, S, V), C).

% -------------------------------------------------
% step : Construct
% -------------------------------------
% Add all the existing nodes from x
hasc((X, S, V), C) :-
% context(C), relate(C,X),
ins(C),
has(X, S, V).

% Add the edge from x to the most specific nodes
hasc((X, S, V), C) :-
% context(C), context(F),
edges_s((X, S, V), C, (Xi, F)).

% Add all other nodes in G(V) in G*(F) if (X,S,V) is added through
   edges_s( )
hasc((Z1, SZ, Z2), C) :-
% context(C), context(F),
edges_s((X, S, V), C, (Xi, F)),
hasc((Z1, SZ, Z2), F),
connect(V, Z2, F),
connect(V, Z1, F).

% -------------------------------------------------
%
% -------------------------------------------------
% ConnectI from X to Y if there is a chain of inheritable edges from
    X to Y
connectI(X, X, C) :- ins(X),
% context(C), relate(C, X),
ins(C).
```

```
connectI(X, Z, C) :-
% context(C),
connectI(X,Y,C),
hasc((Y, S, Z), C),
not noninheritable(S).
% Connect from X to Y if there connectI there is a chain of edges
   from X to Y; the optional last edge is noninheritable.connect(X,
   Y, C) :- connectI(X, Y, C).
connect(X, Y, C) :-
% context(C),
connectI(X,Y,C).
connect(X, Z, C) :-
% context(C),
connectI(X,Y,C),
hasc((Y, S, Z), C),
noninheritable(S).


% ------------------------------------------------
% Define context
% ------------------------------------------------
context(Y) :-     context(X), g_cloned_from(X,Y).

% % Only nodes in G(b1) are related to the context b1
relateI(X, X) :- context(X).
relateI(X, Z) :-
% context(X),
relateI(X,Y),
has(Y, S, Z),
not noninheritable(S).

relate(X,Y) :-
% context(X),
relateI(X,Y).
relate(X, Z) :-
% context(X),
relateI(X,Y),
has(Y, S, Z),
noninheritable(S).

% #show context/1.

% #show class/1.
% #show ins/1.

% #show tc_instance_of/2.

% #show subsume/2.
% #show not_subsume/2.

% #show g_cloned_from/2.

% #show microclones/2.

% #show may_have/3.

% #show may_not_have/2.
```

```
% #show equ/5.

% #show may_not_have2/2.

% #show edges_s/3.

% #show connectI/3.
% #show connect/3.

% #show hasc/2.
```

B.3 Program Π' (Rules - with Modification for Efficiency)

```
noninheritable(prototype_participants;
prototype_participant_of;
prototype_scope;
prototype_of;
cloned_from;
clone_built_from;
instance_of;
clean_instance_of;
big_nodes).

% grab all instances
% class(C) :- has(X, instance_of, C).
ins(X) :- has(X,S,V),
context(C), relate(C, X),
not class(X).
ins(V) :- has(X,S,V),
context(C), relate(C, X),
not class(V).

% ------------------------------------------------
% step 0:
% ------------------------------------------------
instance_of(X,Y) :-    has(X,instance_of,Y).
instance_of(X,Y) :-    has(X,clean_instance_of,Y).
tc_instance_of(X,Y) :- instance_of(X,Y).
tc_instance_of(X,Y) :- tc_instance_of(X,M),  has(M, superclass, Y).

% ------------------------------------------------
% step 1: decide - what clones from what
% ------------------------------------------------
g_cloned_from(X,Y) :- X!=Y, has(X,cloned_from,Y).
g_cloned_from(X,Y) :- X!=Y, has(X,clone_built_from,Y).
g_cloned_from(X,Y) :- X!=Y, tc_instance_of(X,M), has(Y,prototype_of,
   M).


% ------------------------------------------------
% step 2: Sumsume. V1 subsumes V2 if V1 is more specific.
% ------------------------------------------------
% V1 does not subsume V2 if there exist a class C that V2 is tc-
   instance-of but V1 is not.
not_subsume(V1,V2) :-
ins(V1), ins(V2),
```

```
tc_instance_of(V2,C),
not tc_instance_of(V1,C).

subsume(V1,V2) :-
ins(V1), ins(V2),
not not_subsume(V1,V2).

% ------------------------------------------------
% step : Define microclones
% ---------------------------------------
% We let V1 microclones from V2 if V1 subsumes it, so later V1 gets
    expanded.
% microclones((X1,C), (X2,F)) reads as X1 in C is microcloned from
    X2 in F.
% Case 1:
microclones((X2,X2), (X1,X1)) :-
context(X1), context(X2),
ins(X1), ins(X2),
g_cloned_from(X2,X1).

% Case 2:
microclones((V2,C), (V1,F)) :-
context(C), context(F),
ins(X1), ins(X2),
ins(V1), ins(V2),
microclones((X2,C), (X1,F)),
has(X1, S, V1),
has(X2, S, V2),
not noninheritable(S),
subsume(V1,V2).

% Case 3:
microclones((V2,C), (V1,F1)) :-
context(C), context(F1), context(F2),
ins(X1), ins(X2),
ins(V1), ins(V2),
microclones((X,C), (X1,F1)),
microclones((X,C), (X2,F2)),
has(X1, S, V1),
has(X2, S, V2),
X1 != X2,
not noninheritable(S),
subsume(V1,V2).


% % Define edges^*_C(x)
% % ---------------------------------------
% % may_have(X, S, V, C) reads as (X, S, V) is in edges^*_C(x).

% X may have value V from Y if: X microclones value V from some
    instance Y
% may_have((X,S,V),C,(Y,F)): we may have (X,S,V) in C from Y in F
may_have((X,S,V),C,(Y,F)) :-
context(C), context(F),
hasc((Y,S,V),F),
microclones((X,C), (Y,F)),
not noninheritable(S).
```

```
% Result of set difference with respect to subsume: KB has the edge
   (X, S, V1), may have (X, S, V2) from Xj but V1 subsume V2.
% V is in N \_{subsume} M if may_have ((X,S,V), C,(Y,F)) and not
   may_not_have((X, S, V), C)
may_not_have((X, S, V2), C) :-
context(C), context(F),
has(X, S, V1), may_have((X, S, V2), C, (Xj, F)),
subsume(V1, V2),
X != Xj,
V1 != V2.

% Select the most specific node(s)
% -------------------------------------
% Equally specific nodes
equ((X, S, V1), (Xi, Fi), (X, S, V2), (Xj, Fj), C) :-
may_have((X, S, V1), C, (Xi, Fi)), may_have((X, S, V2), C, (Xj, Fj))
   ,
subsume(V2, V1), subsume(V1, V2),
not may_not_have((X, S, V1), C), not may_not_have((X, S, V2), C),
Xi != X, Xj != X, Xi != Xj.

% Most specific nodes: nodes that are not subsumed by any node,
   except its equally specific ones.
may_not_have2((X, S, V1), C) :-
may_have((X, S, V1), C, (Xi,_x)), may_have((X, S, V2), C, (Xj, _)),
subsume(V2, V1), not_subsume(V1, V2),
V1 != V2, Xi != X, Xj != X.

% Because of the imperfection of AURA KB, we have to remove the
   upperbound of the rule to make it work in some cases.
1 { edges_s((X, S, V1), C, (Xj, Fj)): equ((X, S, V), (Xi, Fi), (X, S
   , V1), (Xj, Fj), C);
edges_s((X, S, V), C, (Xi, Fi))} 2 :-
may_have((X, S, V), C, (Xi, Fi)),
not may_not_have((X, S, V), C),
not may_not_have2((X, S, V), C).

% -----------------------------------------------
% step : Construct
% -------------------------------------
% Add all the existing nodes from x
hasc((X, S, V), C) :-
context(C), relate(C,X),
ins(C),
has(X, S, V).

% Add the edge from x to the most specific nodes
hasc((X, S, V), C) :-
context(C), context(F),
edges_s((X, S, V), C, (Xi, F)).

% Add all other nodes in G(V) in G*(F) if (X,S,V) is added through
   edges_s( )
hasc((Z1, SZ, Z2), C) :-
context(C), context(F),
edges_s((X, S, V), C, (Xi, F)),
```

```
hasc((Z1, SZ, Z2), F),
connect(V, Z2, F),
connect(V, Z1, F).

% ------------------------------------------------
%
% ------------------------------------------------
% ConnectI from X to Y if there is a chain of inheritable edges from
    X to Y
connectI(X, X, C) :- ins(X),
context(C), relate(C, X),
ins(C).
connectI(X, Z, C) :-
context(C),
connectI(X,Y,C),
hasc((Y, S, Z), C),
not noninheritable(S).
% Connect from X to Y if there connectI there is a chain of edges
   from X to Y; the optional last edge is noninheritable.connect(X,
   Y, C) :- connectI(X, Y, C).
connect(X, Y, C) :-
context(C),
connectI(X,Y,C).
connect(X, Z, C) :-
context(C),
connectI(X,Y,C),
hasc((Y, S, Z), C),
noninheritable(S).


% ------------------------------------------------
% Define context
% ------------------------------------------------
context(Y) :-     context(X), g_cloned_from(X,Y).

% % Only nodes in G(b1) are related to the context b1
relateI(X, X) :- context(X).
relateI(X, Z) :-
context(X),
relateI(X,Y),
has(Y, S, Z),
not noninheritable(S).

relate(X,Y) :-
context(X),
relateI(X,Y).
relate(X, Z) :-
context(X),
relateI(X,Y),
has(Y, S, Z),
noninheritable(S).

% #show context/1.

% #show class/1.
% #show ins/1.
```

```
% #show tc_instance_of/2.

% #show subsume/2.
% #show not_subsume/2.

% #show g_cloned_from/2.

% #show microclones/2.

% #show may_have/3.

% #show may_not_have/2.

% #show equ/5.

% #show may_not_have2/2.

% #show edges_s/3.

% #show connectI/3.
% #show connect/3.

% #show hasc/2.
```

APPENDIX C

FOR CHAPTER 5

Proof 20 (Corollary 4) *Let x and y be two nodes in the $KDG(z)$. We will prove that there exists an ordering edge from x to y iff all answer sets of $\Pi_G(z) \cup \Pi_{path}$ contain $doconnects(x,y)$. Cases of $dcconnects(x,y)$, $dpconnects(x,y)$, $dlconnects(x,y)$, and $dclconnects(x,y)$ can be proved similarly. We prove this in two directions:*

- ***Forward direction:*** *If there exists an ordering edge from x to y , all answer sets of $\Pi_G(z) \cup \Pi_{path}$ contain $doconnects(x,y)$.*
- ***Backward direction:*** *If an answer set of $\Pi_G(z) \cup \Pi_{path}$ contains $doconnects(x,y)$, there exists an ordering edge from x to y.*

Forward direction: *If there exists an ordering edge from x to y , all answer sets of $\Pi_G(z) \cup \Pi_{path}$ contain $doconnects(x,y)$.*

1. *Our knowledge base representation makes sure that*
   (a) *Both of x and y are events: $event(x)$ and $event(y)$ are presented in all answer sets (of the program $\Pi_G(z) \cup \Pi_{path}$)*
   (b) *$has(X,S,Y)$ must be true where S is one of the predicates for compositional edges defined in rule e2.*
2. *Rule e7, then, makes sure all answer sets containing $doconnects(x,y)$.*

Backward direction: *If an answer set of $\Pi_G(z) \cup \Pi_{path}$ contains $doconnects(x,y)$, there exists an ordering edge from x to y.*

1. *If the answer set contains $doconnects(x,y)$, it must be introduced by rule e7.*
2. *Both x and y must be events and there exists an edge of type S where $ordering\_edge(S)$ is true.*
3. *Hence, there exists an ordering edge from x to y.*

Proof 21 (Corollary 5) *This can be proved using Corollary 4.*

If there exists a cpath edge from x to y, according to corollary 4, all answer sets contain either $dcconnects(x,y)$, $dpconnects(x,y)$, or $dlconnects(x,y)$. Rules c1, c2 or c3 make sure $dcpath\_connects(x,y)$ is true in all answer sets.

In the backward direction, when $dcpath\_connects(x,y)$ is true, it must be introduced by rule c1, c2 or c3. This means at least one of the three $dcconnects(x,y)$, $dpconnects(x,y)$, and $dlconnects(x,y)$ must be true. Using Corollary 4, we conclude that there exists a cpath edge from x to y.

Proof 22 (Proposition 7) *If there exists a cpath from x to y, there must be a list of different nodes $N_0, N_1, ..., N_k$ where*

1. *$N_0 = x$*
2. *$N_k = y$, $k > 0$*

3. *There exists a cpath edge from N_i to N_{i+1}, $0 \leq i < k$*

Rule c4 makes $cpath\_connects(N_0,N_1)$ true. Rule c5 makes $cpath\_connects(N_0,N_2)$ true, then $cpath\_connects(N_0,N_3)$ true, then ..., $cpath\_connects(N_0,N_k) = cpath\_connects(x,y)$ true.

In the backward direction, when $cpath\_connects(x,y)$ is true, it must be introduced by rule c4 or c5. We can, then, argue that the answer set contains a set of facts $dcpath\_connects(N_{i+1},N_i)$ where

1. $N_0 = y$
2. $N_k = x$, $k > 0$

This mean there exists a cpath from x to y through $N_{k-1},N_{k-2},...,N_1$.

Proof 23 (Proposition 8) *Let $q1$ be the question "How does xc work?". Let x be a node in the $KDG(z)$ so that xc is x's class. Let K be a $(z,x)\_KDG$. Let minnumstep be the length of the cpath from z to x in K. We are going to prove that here exists an answer set of $\Pi^1(q1,z,numstep)$ containing:*

$$ans(q1,zx\_kdg(q1,z,x),node,N)$$

and

$$ans(q1,zx\_kdg(q1,z,x),edge,has(I,Predicate,J))$$

where $numstep \geq minnumstep$; N is any node of K, and $(I,Predicate,J)$ is any edge of K.

- *By splitting program $\Pi^1(q,z,numstep) = \Pi_G(z) \cup \Pi_{path} \cup \Pi_Q(q) \cup \Pi^1_{answer}(numstep)$ to two strata, $\Pi_G(z) \cup \Pi_{path}$ and $\Pi_Q(q) \cup \Pi^1_{answer}(numstep)$, we have that the answer set of $\Pi_G(z) \cup \Pi_{path}$ is included in a answer set of $\Pi^1(q,z,numstep)$. We thus still can use Corollary 4, Corollary 5 and Proposition 7 with $\Pi^1(q,z,numstep)$*
- *Since there exists a cpath from z to x, all answer sets of $\Pi^1(q,z,numstep)$ must have $cpath\_connects(z,x)$.*
- *Since x is an instance of XClass, $\Pi^1(q,z,numstep)$ must have $has(x,instance\_of,xc)$.*
- *Encoding of the question $q1$ $\Pi_Q(q1)$ will be*

```
question(q1).
has(q1, type, how).
has(q1, category, work).
has(q1, param1, xc).
```

- *Rule $z1$ introduces $zx\_kdg(q1,z,x)$ to all answer sets of $\Pi^1(q,z,numstep)$.*
- *Let $N_0,N_1,...,N_{minnumstep}$ to be the nodes on the cpath from z to x of K, $N_0 = z$, $N_{minnumstep} = x$. Rule $p1$ gives $selected\_path1(z,x,z,0)$ = $selected\_path1(N_0,N_{minnumstep},N_0,0)$, to all answer sets.*

- *Rule p2 makes sure at least one answer set containing* $selected\_path1(z,x,N_1,1)$, *then at least one answer set containing both* $selected\_path1(z,x,N_1,1)$ *and* $selected\_path1(z,x,N_2,2)$, *then at least one answer set containing all* $selected\_path1(z,x,N_1,1)$, $selected\_path1(z,x,N_2,2)$, *and* $selected\_path1(z,x,N_3,3)$. ...

- *There will be at least one answer set containing all* $selected\_path1(z,x,N_1,1)$, ...,$selected\_path1(z,x,N_i,i)$, ..., $selected\_path1(z,x,N_{minnumstep},minnumstep)$. *Let that answer set be ANS. Note that* $numstep \geq minnumstep$.

- *ANS will contain* $completed\_path1(zx\_kdg(q1,z,x))$. *The body of rule p4 thus will not be satisfied.*

- *For any node N of K, it is either in the cpath from z to x or in* $KDG(x)$. *In the first case, ANS will contain* $selected\_path1(z,x,N,i), 0 \leq i \leq minnumstep$. *In the second case, there must be a cpath from x to N, ANS will contain* $cpath\_connects(x,N)$.

- *Rule a1 and a2 will make sure that ANS contains* $ans(q1,zx\_kdg(q1,z,x),node,N)$.

- *Rule a3 ,then, will make sure that ANS contains* $ans(q1,zx\_kdg(q1,z,x),edge,has(I,Predicate,J))$, *with* $(I,Predicate,J)$ *is any edge of K.*

Proof 24 (Proposition 9) *Let q1 be the question "How does xc work?". Let x be a node in the* $KDG(z)$ *so that xc is x's class. Suppose there exists at least one* $(z,x)\_KDG$ *where the length of cpath from z to x is less than or equal numstep. Let A be an answer set of* $\Pi^1(q1,z,numstep)$, *let* $V = \{n : ans(q1,zx\_kdg(q1,z,x),node,n) \in A\}$ *and* $E = \{(i,Predicate,j) : ans(q1,zx\_kdg(q1,z,x),edge,has(i,Predicate,j)) \in A\}$.

We are going to prove that there exists a $(z,x)\_KDG$ *K so that V is K's set of nodes and E is K's set of edges. We will first point out the* $(z,x)\_KDG$ *K that contains the set of nodes V and set of edges E. Later we will prove that K does not contains any more node (not in V).*

1. *From rule a1 or a2, we know that A must contain* $zx\_kdg(q1,z,x)$, *and either* $cpath\_connects(x,N)$ *or* $selected\_path1(z,x,N,i)$.

2. *From rule p4, we know that A also contain* $completed\_path1(zx\_kdg(q1,z,x))$.

3. *p3 then gives us that A contains* $selected\_path1(z,x,x,t)$, $t \leq numstep$. *p1 gives us that A contains* $selected\_path1(z,x,z,0)$.

4. *Let* $N_0 = x$, *rule p2 shows that A must contains*

   (a) $selected\_path1(z,x,N_1,t-1)$

   (b) $dcpath\_connects(N_1,N_0)$

   (c) $cpath\_connects2(N_1,z)$

This means that there exists a cpath from N_1 to z, and a cpath edge from N_1 to N_0.

5. *Use previous step repeatedly, we have a list of nodes $N_0, N_1, ..., N_t$, where $N_0 = x, N_t = z$, that forms a cpath from z to x. Let K be the (z,x)_KDG forms by that cpath and $KDG(x)$. Similar to previous proof, we can prove that A must contain $ans(q1, zx\_kdg(q1, z, x), node, N)$ where N is any node in $KDG(x)$.*

6. *V thus contains all the nodes of K.*

7. *Now let us assume that V contains node N' that is not in K. N' must be introduced by either rule a1 or a2.*

8. *A contains either $cpath\_connects(x, N')$ or $selected\_path1(z, x, N', i)$.*

9. *If A contains $cpath\_connects(x, N')$, there must exist a cpath from x to N'. N' must be in $KDG(x)$, contradiction.*

10. *If A contains $selected\_path1(z, x, N', i)$, similarly to previous steps, we can prove that there exists a list of node $N'_0, N'_1, ..., N'_i$ that forms a cpath from z to N' ($N'_0 = N'$, $N'_i = z$). This cpath and the cpath formed by $N_0, N_1, ..., N_t$ all start at z, so they must share some nodes and fork some node $N_k = N'_{k'}$. This means their next nodes N_{k-1} and $N'_{k'-1}$ are different from each other.*

11. *A, then, contains both $selected\_path1(z, x, N_{k-1}, l+1)$ and $selected\_path1(z, x, N'_{k'-1}, l+1)$, where l is the length of cpath from z to $N_k = N'_{k'}$. This is contradict to the upper-bound limit (= 1) of rule p2.*

12. *V thus contains no such node N', so contains no edge that is not in E.*

13. *V is thus K's set of nodes and E is K's set of edges.*

Propositions 10 to 17 can be proved similarly to Propositions 8 and 9 above.

APPENDIX D

FOR CHAPTER 6

ID	Sentence	Meaning
1	List jobs in loc1	λ x.answer(x,job(x) $\wedge$ loc(x,loc1))
2	List jobs in area1	λ x.answer(x,job(x) $\wedge$ area(x,area1))
3	Show jobs using language1	λ x.answer(x,job(x) $\wedge$ language(x,language1))
4	List jobs requiring reqdeg1	λ x.answer(x,job(x) $\wedge$ req_deg(x,reqdeg1))
5	List jobs requiring reqdeg1 using language1	λ x.answer(x,job(x) $\wedge$ req_deg(x,reqdeg1) $\wedge$ language(x,language1))
6	Are there any jobs in loc1	λ x.answer(x,job(x) $\wedge$ loc(x,loc1))
7	Are there any jobs specializing in area1 with company1	λ x.answer(x,job(x) $\wedge$ area(x,area1) $\wedge$ company(x,company1))

Table D.1: Sentences and Their Meanings in JobsCompact7 Corpus

D.1 Training on JobsCompact7 Corpus

In this section, we will show details of how the learning algorithm works in each iteration. We demonstrate step-by-step of the learning process on a small corpus, which is extracted from Jobs corpus.

Training Sentences and CCG Parse Trees We start with JobsCompact7, its sentences, and their translations shown in Table D.1. In the real sentences in Jobs corpus, there are entities like "Java" "C++." We pre-processed the sentences with named entity recognizers and replaced them by their tag. For example, "Java" is replaced by "language1" (or language2 if language1 has already appeared in the same sentence). JobsCompact7 contains only preprocessed sentences. While preprocessing, we keep the list of replaced words/phrases for each sentences so that we can replace "language1" or "company1" to the original word later. Please note that we actually use syntax II (please see section 6.3.1) in the experiments, but here we show λ-expressions (in figures) in the form similar to syntax I only for display purposes.

First of all, we check the CCG parse trees of all the sentences in the corpus. Their CCG parsing trees are shown in D.1 - D.18

Initial Dictionary As we can see in the CCG trees, the words "List" or "Show" are always the child of the root node (S); they combine with the rest of the sentence to make the full sentence. Moreover, from Table D.1, "List" or "Show" seems to have the meaning $\lambda x.answer(x,...)$. We construct the initial dictionary in Table D.2, where the word's CCG categories are from the CCG parse trees. We will explain the meaning of each word and why we chose it.

- The meaning of "List" is λ p. λ x.answer(x,p @ x), which means that we are answering something, and we are expecting a lambda expression p to explain more about what we are answering. The first unbound variable (u if p has the form $\lambda u.()$) in p will be replaced by x.
- "jobs" simply means λ x.job(x).

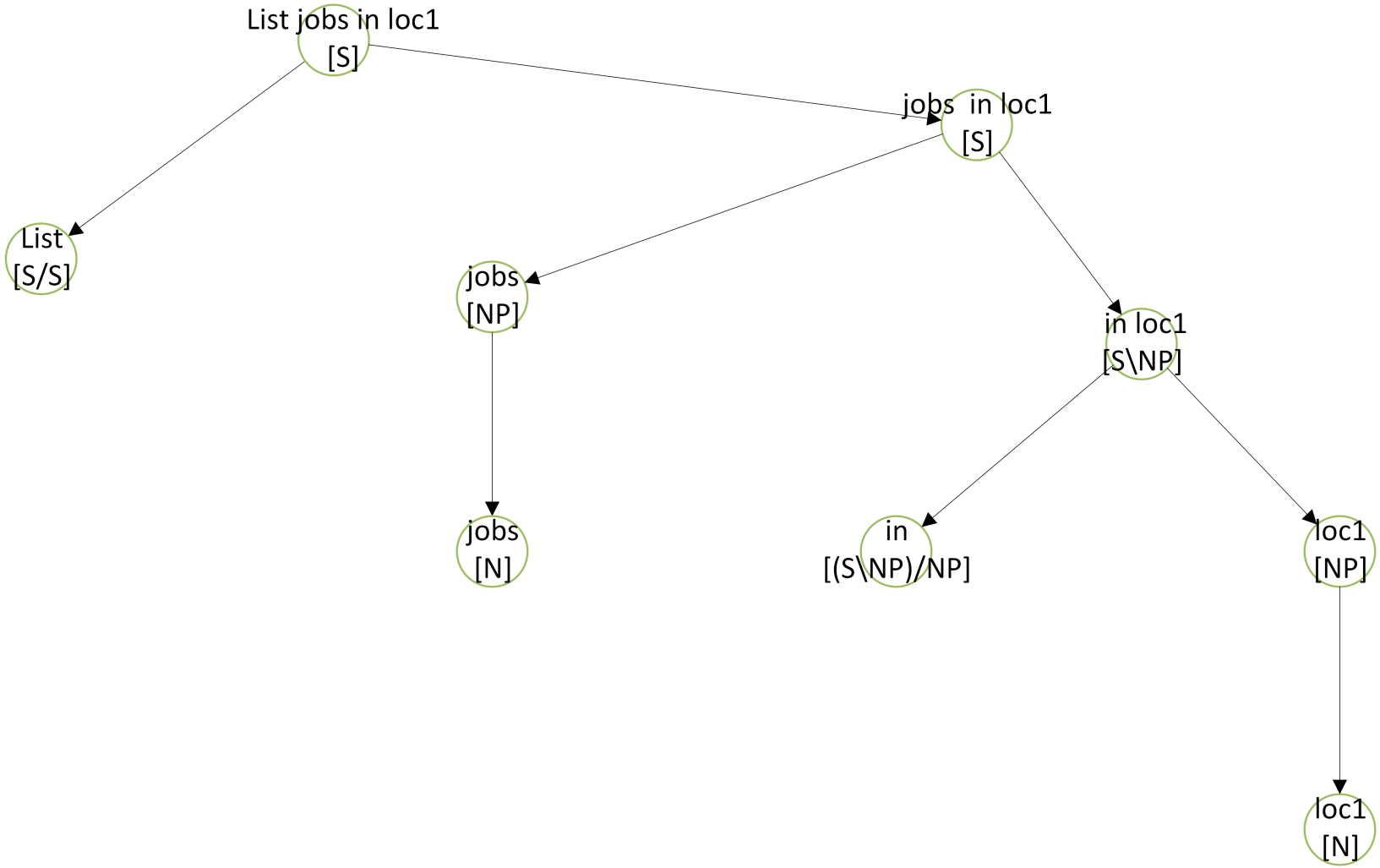

Figure D.1: CCG parse tree of sentence 1

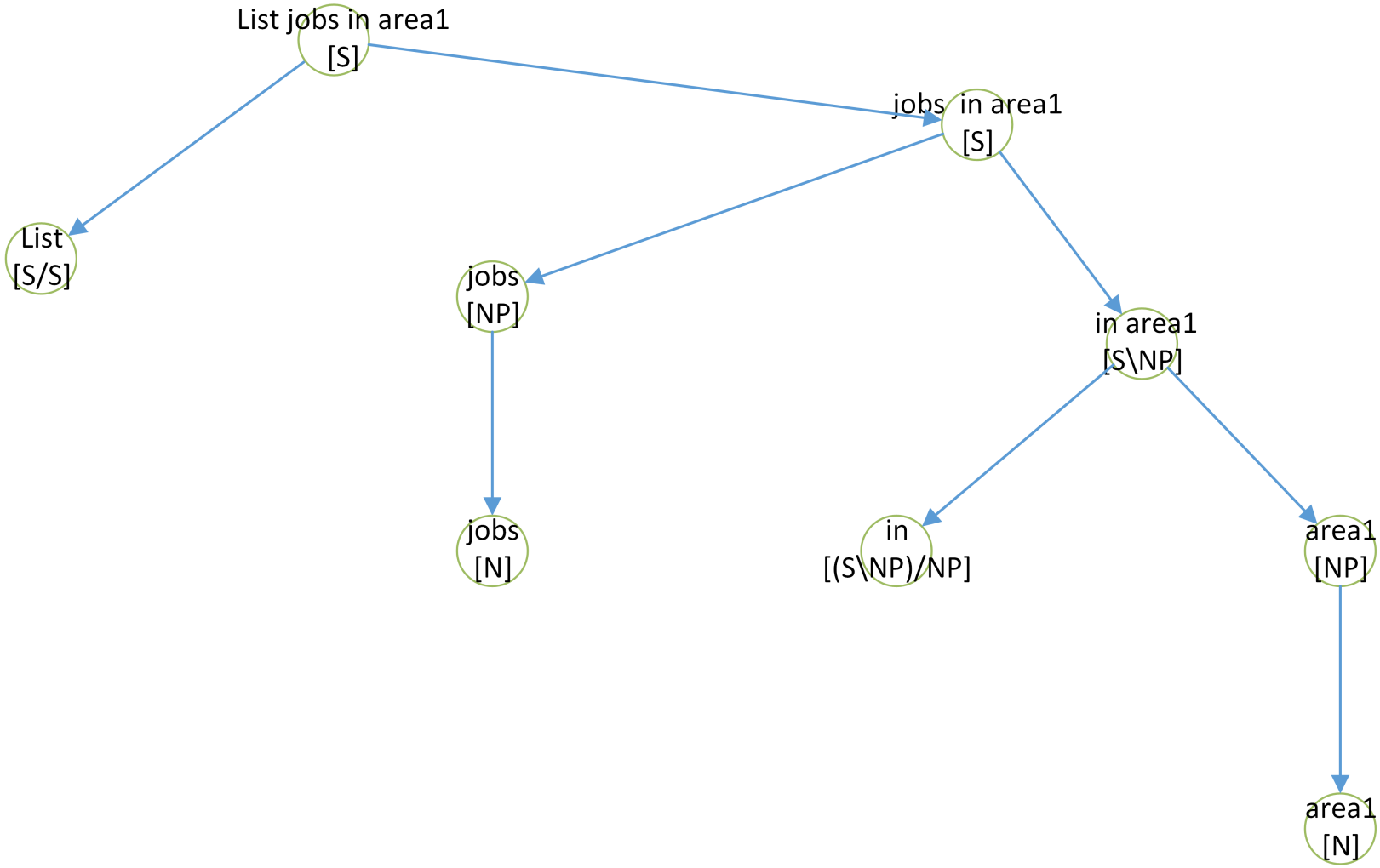

Figure D.2: CCG parse tree of sentence 2

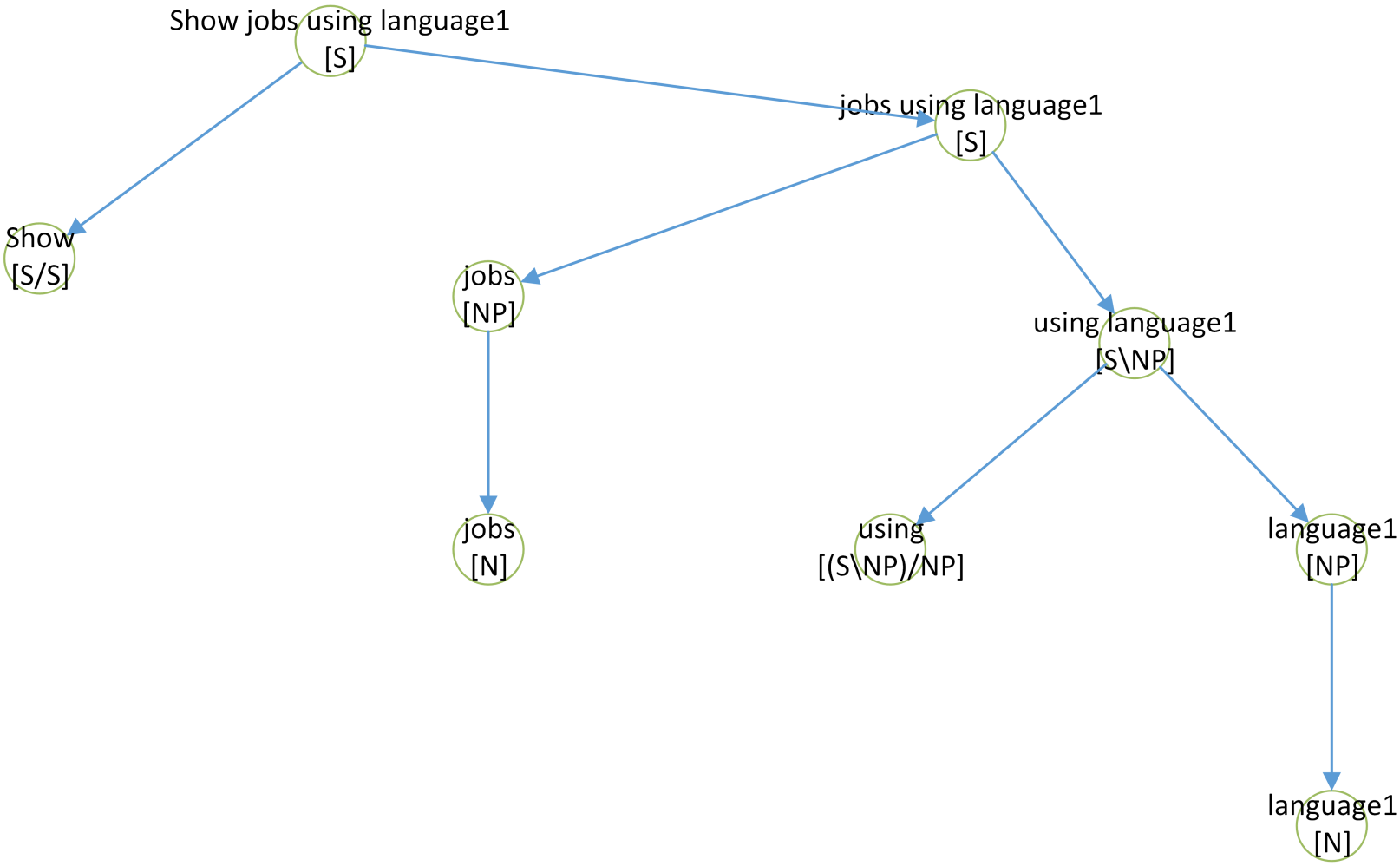

Figure D.3: CCG parse tree of sentence 3

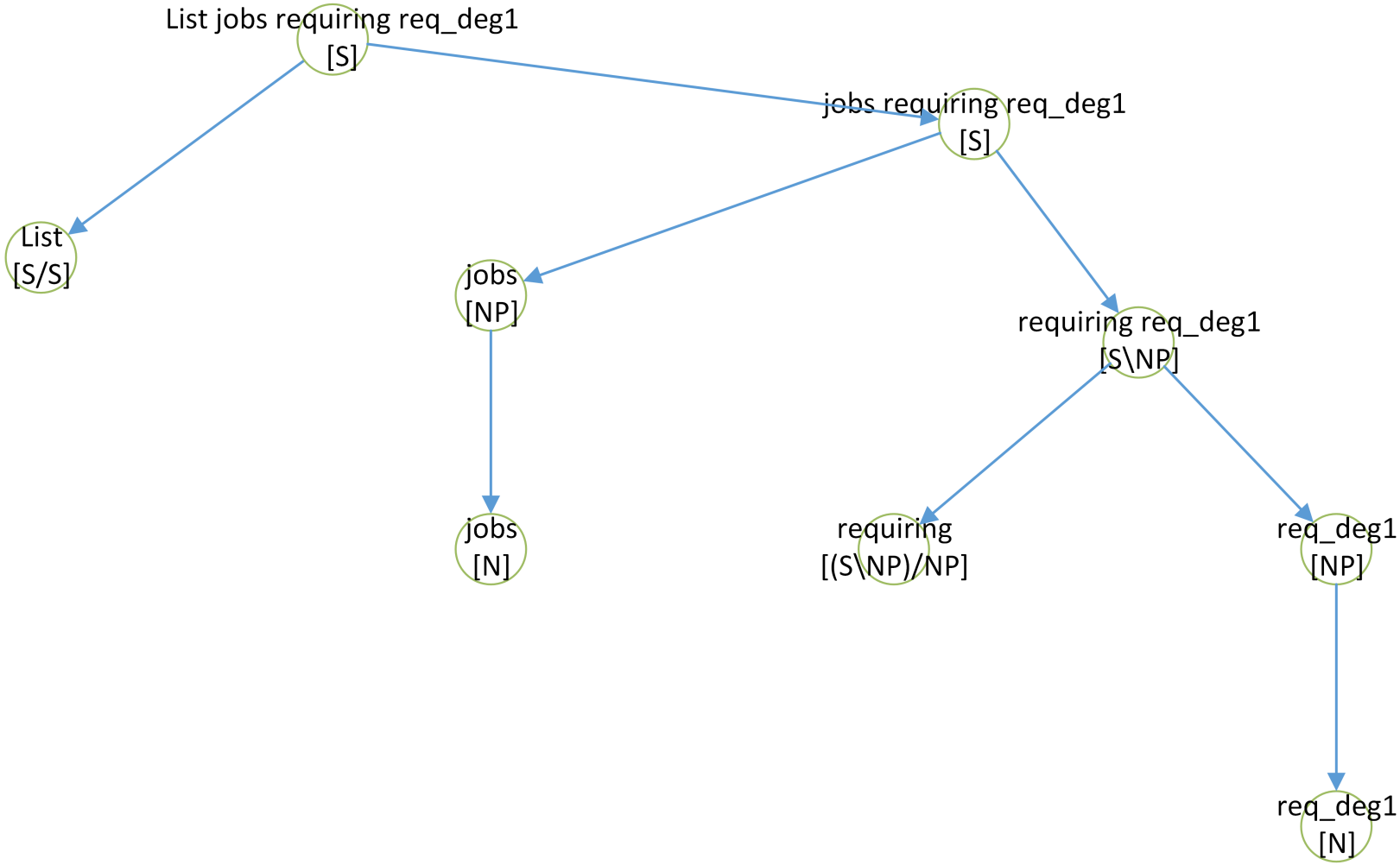

Figure D.4: CCG parse tree of sentence 4

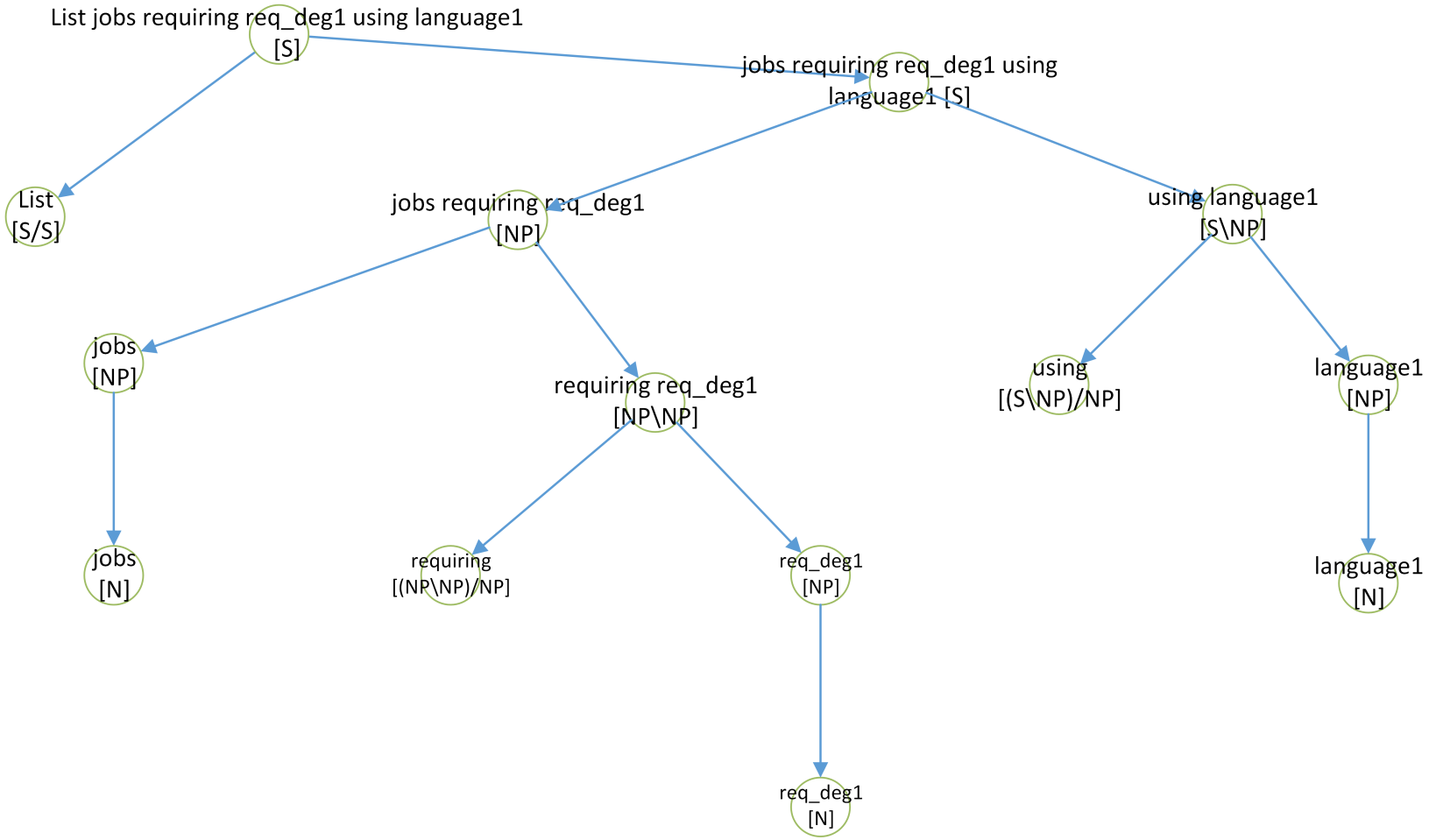

Figure D.5: CCG parse tree of sentence 5

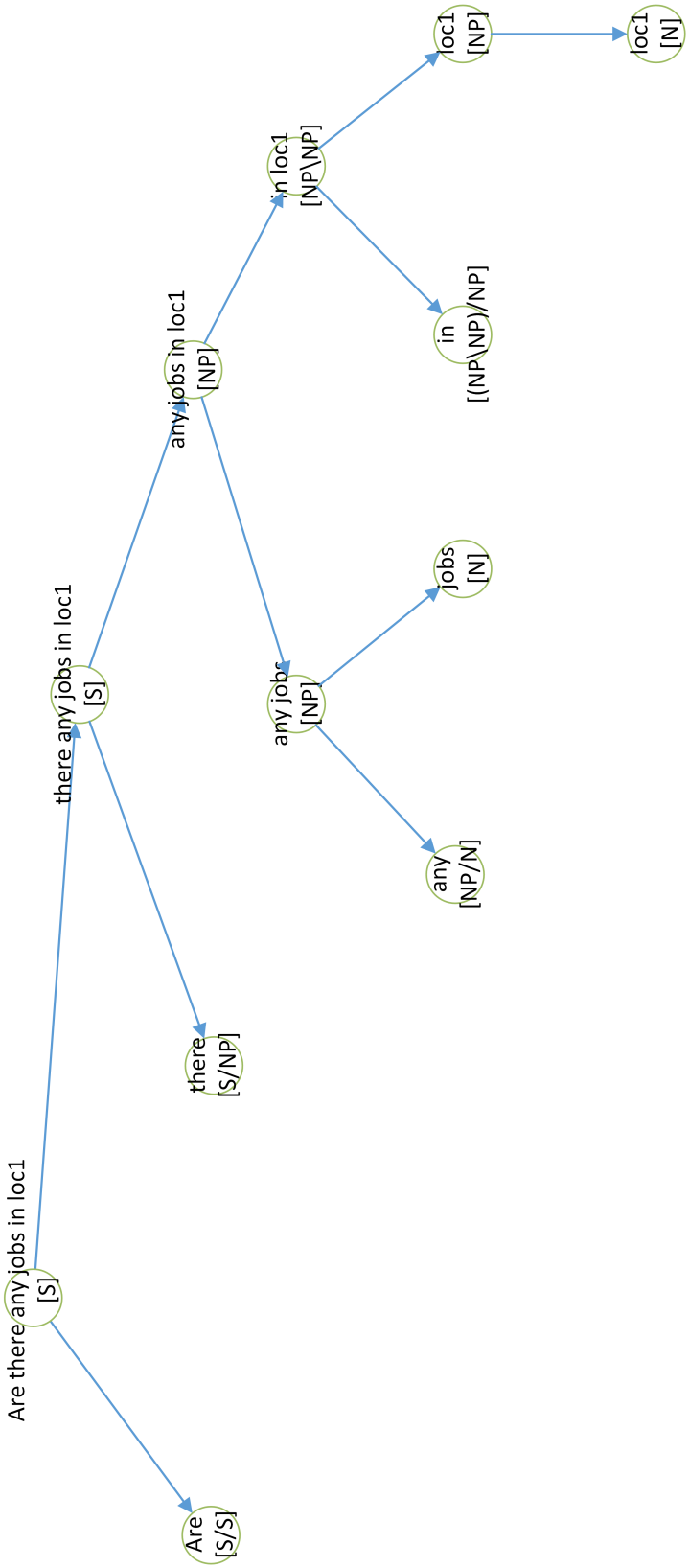

Figure D.6: CCG parse tree of sentence 6

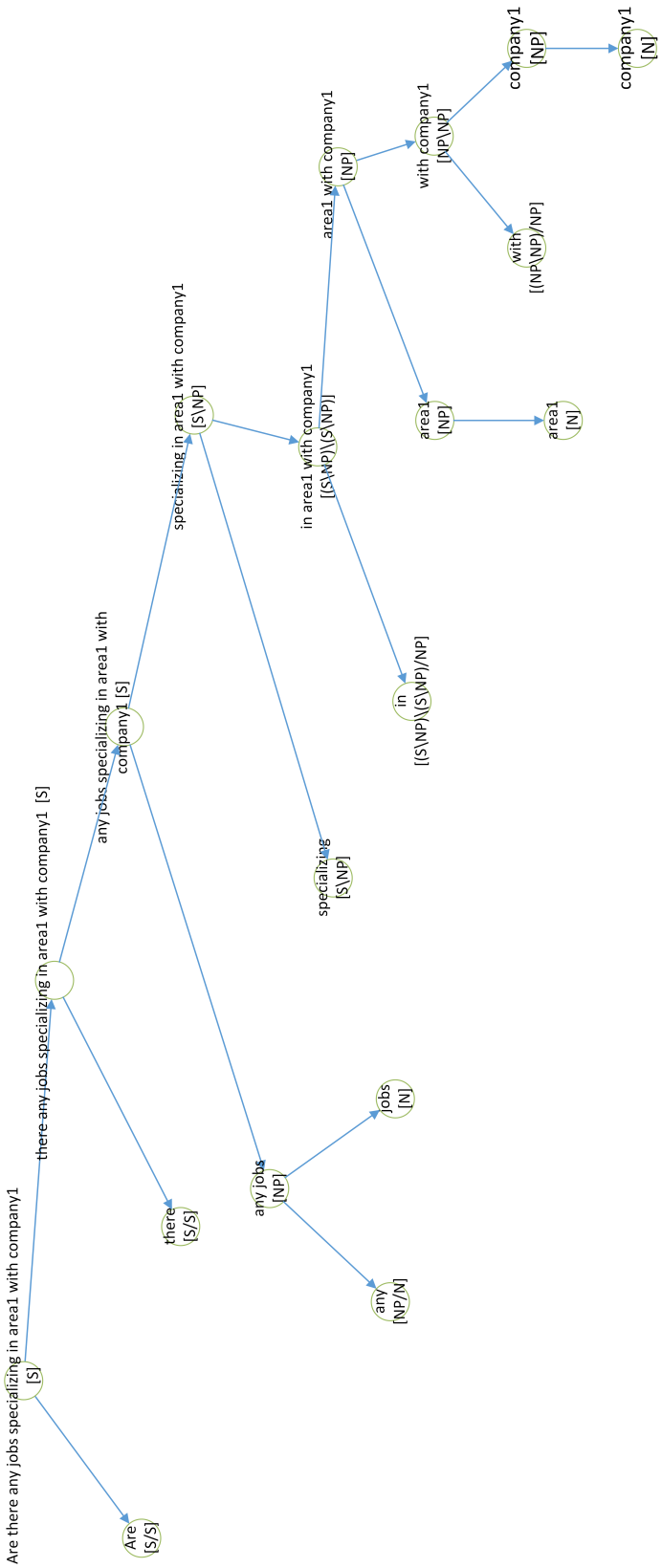

Figure D.7: CCG parse tree of sentence 7

Word	CCG	Meaning
List	S/S	λ p. λ x.answer(x,p @ x)
jobs	N	λ x.job(x)
area1	N	λ x.area(x,area1)
company1	N	λ x.company(x,company1)
language1	N	λ x.language(x,language1)
loc1	N	λ x.loc(x,loc1)
reqdeg1	N	λ x.req_deg(x,reqdeg1)
least	N	λ x.x

Table D.2: Initial Dictionary for JobsCompact Corpus

- "area1", "company1" or "language1"... means λ x.area(x,area1), λ x.company(x, company1) or λ x.language(x, language1). These meanings can be generated automatically by a small script from our list of entities: area, company, language, location, etc.
- "least" means λ x.x, which tells that "least" does not have any particular meanings in the translation; whatever combines with it will keep its meaning. Giving this trivial meaning of "least" is optional since the learning algorithm can generalize this kind of meaning later.

Iteration 0: In the following is the log of the NL2KR training on JobsCompact7 corpus. As we can see, the lambda expressions are represented in the representation mentioned in previous section; $\lambda x.job(x)$ is represented by $l(x, f(job, x))$.

```
INFO: ******Reading lexicon file: C:\Users\Nguyen\Documents\
   workspace\NL2KR\PCCG_New\corpora\6sentences\dict.txt
INFO: Parsing line 0: List    S/S   l(p, l(x, f(answer, a(p,x))))
INFO: Skipping line 1: //requiring  (NP\NP)/NP  l(p, l(q, l(x, g(and
   , a(q,x), a(p,x)))))
INFO: Parsing line 2: jobs    N     l(x, f(job,x))
INFO: Parsing line 3: area1   N     l(x, f(area,x,area1))
INFO: Parsing line 4: company1      N     l(x, f(company,x,company1)
   )
INFO: Parsing line 5: language1     N     l(x, f(language,x,
   language1))
INFO: Parsing line 6: loc1    N     l(x, f(loc,x,loc1))
INFO: Parsing line 7: reqdeg1 N     l(x, f(reqdeg,x,reqdeg1))
INFO: Parsing line 8: least   N     l(x,x)
INFO: Skipping line 9: //using      (S\NP)/S    l(p, l(q, l(x, g(and
   , a(q,x), a(p,x)))))
INFO: LearningProcess: 8 initial lexicon read.
INFO: ******Reading training file: C:\Users\Nguyen\Documents\
   workspace\NL2KR\PCCG_New\corpora\6sentences\train.txt
INFO: Parsing line 0: List jobs in loc1   l(x,f(answer,g(and, f(job,
    x), f(loc, x, loc1))))
INFO: Parsing line 1: List jobs in area1  l(x,f(answer,g(and, f(job,
    x), f(area, x, area1))))
INFO: Parsing line 2: Show jobs using language1       l(x,f(answer,g
   (and, f(job, x), f(language, x, language1))))
```

```
INFO: Parsing line 3: List jobs requiring reqdeg1     l(x,f(answer,g
   (and, f(job, x), f(reqdeg, x, reqdeg1))))
INFO: Parsing line 4: List jobs requiring reqdeg1 using language1
         l(x,f(answer,g(and, f(job, x), f(reqdeg, x, reqdeg1), f(
   language, x, language1))))
INFO: Parsing line 5: Are there any jobs in loc1      l(x,f(answer,g
   (and, f(job, x), f(loc, x, loc1))))
INFO: Parsing line 6: Are there any jobs specializing in area1 with
   company1  l(x,f(answer,g(and, f(job, x), f(area, x, area1), f(
   company, x, company1))))
INFO: Skipping line 7: //Are there any jobs requiring a reqdeg1 for
   company1 in loc1      l(x,f(answer,g(and, f(job, x), f(reqdeg, x,
    reqdeg1), f(company, x, company1), f(loc, x, loc1))))
INFO: Skipping line 8: //Are there any jobs in loc1 requiring at
   least a reqdeg1 and knowing language1      l(x,f(answer,g(and, f(
   job, x), f(loc, x, loc1), f(reqdeg, x, reqdeg1), f(language, x,
   language1))))
INFO: LearningProcess: 7 training data read.
INFO: ******Learning lexicon ...
```

Iteration 1: In the following is the log of learning iteration 1. Let us look more carefully at the first 5 lines where the sentence 1 (List jobs in loc1) is learned.

```
INFO: Trying to learn 'S' [List jobs in loc1]: expected = {1} [l(x,
   f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)], current =
   {0} []
INFO: Trying to learn 'S' [jobs in loc1]: expected = {1} [l(x, g(and
   , f(job, x), f(loc, x, loc1)))(0.0)], current = {0} []
INFO: Trying to learn 'S\NP' [in loc1]: expected = {1} [l(x0, l(x, g
   (and, a(x0, x), f(loc, x, loc1))))(0.0)], current = {0} []
INFO: Trying to learn '(S\NP)/NP' [in]: expected = {1} [l(x1, l(x0,
   l(x, g(and, a(x0, x), a(x1, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [List jobs in area1]: expected = {1} [l(x,
    f(answer, g(and, f(job, x), f(area, x, area1))))(0.0)], current
   = {0} []
INFO: Trying to learn 'S' [jobs in area1]: expected = {1} [l(x, g(
   and, f(job, x), f(area, x, area1)))(0.0)], current = {0} []
INFO: Trying to learn 'S\NP' [in area1]: expected = {1} [l(x0, l(x,
   g(and, a(x0, x), f(area, x, area1))))(0.0)], current = {0} []
INFO: Trying to learn '(S\NP)/NP' [in]: expected = {1} [l(x1, l(x0,
   l(x, g(and, a(x0, x), a(x1, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [Show jobs using language1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(language, x, language1)))
   )(0.0)], current = {0} []
INFO: Trying to learn 'S' [List jobs requiring reqdeg1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(reqdeg, x, reqdeg1))))
   (0.0)], current = {0} []
INFO: Trying to learn 'S' [jobs requiring reqdeg1]: expected = {1} [
   l(x, g(and, f(job, x), f(reqdeg, x, reqdeg1)))(0.0)], current =
   {0} []
INFO: Trying to learn 'S\NP' [requiring reqdeg1]: expected = {1} [l(
   x0, l(x, g(and, a(x0, x), f(reqdeg, x, reqdeg1))))(0.0)], current
    = {0} []
INFO: Trying to learn '(S\NP)/NP' [requiring]: expected = {1} [l(x1,
    l(x0, l(x, g(and, a(x0, x), a(x1, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [List jobs requiring reqdeg1 using
   language1]: expected = {1} [l(x, f(answer, g(and, f(job, x), f(
   reqdeg, x, reqdeg1), f(language, x, language1))))(0.0)], current
```

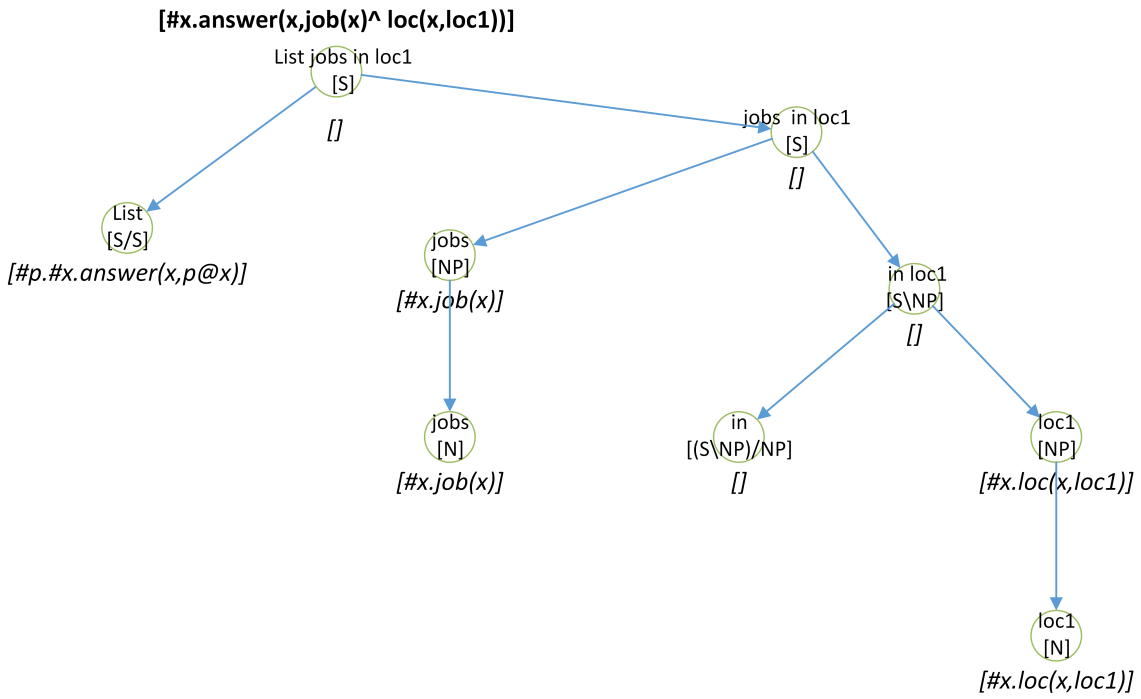

Figure D.8: Current status of learning sentence 1 in iteration 0

```
   = {0} []
INFO: Trying to learn 'S' [jobs requiring reqdeg1 using language1]:
   expected = {1} [l(x, g(and, f(job, x), f(reqdeg, x, reqdeg1), f(
   language, x, language1)))(0.0)], current = {0} []
INFO: Trying to learn 'S' [Are there any jobs in loc1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)],
   current = {0} []
INFO: Trying to learn 'S' [Are there any jobs specializing in area1
   with company1]: expected = {1} [l(x, f(answer, g(and, f(job, x),
   f(area, x, area1), f(company, x, company1))))(0.0)], current =
   {0} []
INFO: Learned 'in : (S\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0, x), a(
   x1, x)))))' by inverse
INFO: Learned 'requiring : (S\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0,
    x), a(x1, x)))))' by inverse
INFO: We learned 2 new lexicon(s):
```

Shown in Figure D.8 is what we already known in sentence 1. We have the meanings of "List", "jobs," and "loc1." The current meanings of those words are shown below each leaf node (in italics). The bottom-up process (subsection 6.4.2) combines the current meanings from leaf nodes to root; since the meaning of "in" is unknown, we do not have current meanings of "in loc1", "jobs in loc1," and "List jobs in loc1." However, we do know the expected meaning of "List jobs in loc1" (show on top of root node, in bold).

Shown in Figure D.9 is the top-down process (subsection 6.4.2) to learn expected meanings, which corresponds to the first 5 lines in the log. Using inverse lambda, we use the current meaning of "List" to learn an expected meaning of "jobs in loc1" (shown on top of node "jobs in loc1" in bold). Then, we continue using inverse lambda with the cur-

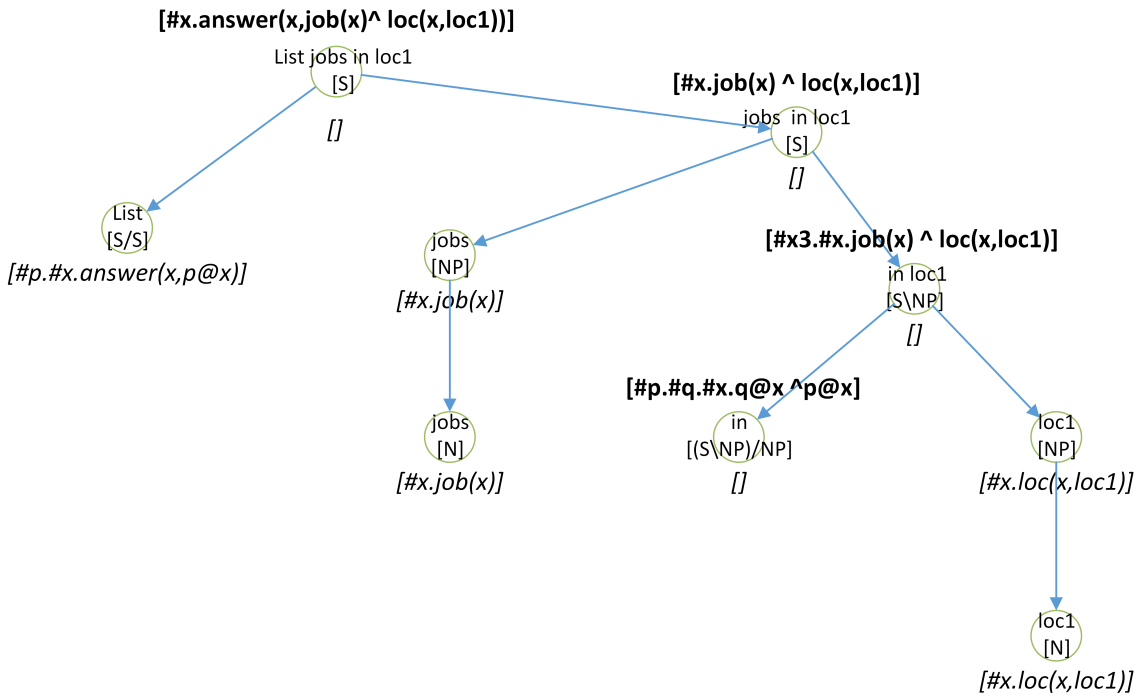

Figure D.9: Current status of learning sentence 1 in iteration 1

rent meaning of "jobs" to get an expected meaning of "in loc1." Similarly, we obtain an expected meaning of "in" λ p. λ q. λ x.q @ x ∧ p @ x.

Here we know the most in 3 sentences:

1. List jobs in loc1
2. List jobs in area1'
3. List jobs requiring reqdeg1

From those sentences, we can learn the meaning of the two words "in" and "requiring." For other sentences, we do not have enough information for the inverse learning to start.

Iteration 2:

```
SEVERE: ============================ Iteration 2
   ============================
INFO: Trying to learn 'S' [List jobs in loc1]: expected = {1} [l(x,
   f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)], current =
   {1} [l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))(-1.0)]
INFO: Trying to learn 'S' [List jobs in area1]: expected = {1} [l(x,
    f(answer, g(and, f(job, x), f(area, x, area1))))(0.0)], current
   = {1} [l(x, f(answer, g(and, f(job, x), f(area, x, area1))))
   (-1.0)]
INFO: Trying to learn 'S' [List jobs requiring reqdeg1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(reqdeg, x, reqdeg1))))
   (0.0)], current = {1} [l(x, f(answer, g(and, f(job, x), f(reqdeg,
    x, reqdeg1))))(-1.0)]
INFO: We learned 0 new lexicon(s):
```

In this iteration, using the meanings that we have learned so far, we got the current meanings of each sentence. The current meanings of the three sentences mentioned above contain their expected meanings. For example, in the first line of the log for iteration 2, for sentence "List jobs in loc1," we have 1 expected meaning
$l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))$ and 1 current meaning:
$l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))$. Since these two meanings are equal, we have finished learning this sentence. In this iteration, we verify that we have finished learning the 3 sentences and learn no new words.

Iteration 3:

```
SEVERE: ============================ Iteration 3
   ============================
INFO: Learned 'in : (S\NP)/NP : l(x, x)' by generalization
INFO: Learned 'requiring : (NP\NP)/NP : l(x, x)' by generalization
INFO: Learned 'Are : S/S : l(p, l(x, f(answer, a(p, x))))' by
   generalization
INFO: Learned 'Are : S/S : l(x, x)' by generalization
INFO: Learned 'any : NP/N : l(x, x)' by generalization
INFO: Learned 'there : S/S : l(x, x)' by generalization
INFO: Learned 'List : S/S : l(x, x)' by generalization
INFO: Learned 'using : (S\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0, x),
    a(x1, x))))' by generalization
INFO: Learned 'using : (S\NP)/NP : l(x, x)' by generalization
INFO: Learned 'specializing : S\NP : l(x, x)' by generalization
INFO: Learned 'in : ((S\NP)\(S\NP))/NP : l(x, x)' by generalization
INFO: Learned 'with : (NP\NP)/NP : l(x, x)' by generalization
INFO: Learned 'Show : S/S : l(p, l(x, f(answer, a(p, x))))' by
   generalization
INFO: Learned 'Show : S/S : l(x, x)' by generalization
INFO: We learned 14 new lexicon(s):
```

Because we cannot learn any new words in the previous iteration, we begin using Generalization. In this iteration, we learn 14 new meanings through generalization. These meanings are generalized from two sources:

1. The meanings we learned by inversion in the previous iteration such as "in" and "requiring." For example, meanings $l(p, l(q, l(x, g(and, a(q, x), a(p, x)))))$ of "for" is from this source.

2. The trivial meanings defined (in trivialTemplates.txt file) which contain $l(x, x)$ (means λ x.x)

Iteration 4:

```
SEVERE: ============================ Iteration 4
   ============================
INFO: Trying to learn 'S' [Show jobs using language1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(language, x, language1)))
   )(0.0)], current = {4} [l(x, f(answer, g(and, f(job, x), f(
   language, x, language1))))(-2.0), l(x, f(answer, a(f(language, l(
   x, f(job, x)), language1), x)))(-2.0), l(x, g(and, f(job, x), f(
   language, x, language1)))(-2.0), f(language, l(x, f(job, x)),
   language1)(-2.0)]
INFO: Trying to learn 'S' [List jobs requiring reqdeg1 using
   language1]: expected = {1} [l(x, f(answer, g(and, f(job, x), f(
```

```
   reqdeg, x, reqdeg1), f(language, x, language1))))(0.0)], current
   = {4} [l(x, f(answer, g(and, a(f(reqdeg, l(x, f(job, x)), reqdeg1
   ), x), f(language, x, language1))))(-2.0), f(language, f(reqdeg,
   l(x, f(job, x)), reqdeg1), language1)(-3.0), l(x, g(and, a(f(
   reqdeg, l(x, f(job, x)), reqdeg1), x), f(language, x, language1))
   )(-3.0), l(x, f(answer, a(f(language, f(reqdeg, l(x, f(job, x)),
   reqdeg1), language1), x)))(-2.0)]
INFO: Trying to learn 'S' [jobs requiring reqdeg1 using language1]:
   expected = {2} [l(x, g(and, f(job, x), f(reqdeg, x, reqdeg1), f(
   language, x, language1)))(0.0), l(x, f(answer, g(and, f(job, x),
   f(reqdeg, x, reqdeg1), f(language, x, language1))))(-1.0)],
   current = {2} [l(x, g(and, a(f(reqdeg, l(x, f(job, x)), reqdeg1),
    x), f(language, x, language1)))(-2.0), f(language, f(reqdeg, l(x
   , f(job, x)), reqdeg1), language1)(-2.0)]
INFO: Trying to learn 'NP' [jobs requiring reqdeg1]: expected = {1}
   [l(x, g(and, f(job, x), f(reqdeg, x, reqdeg1)))(-1.0)], current =
    {1} [f(reqdeg, l(x, f(job, x)), reqdeg1)(-1.0)]
INFO: Trying to learn 'NP\NP' [requiring reqdeg1]: expected = {1} [l
   (x0, l(x, g(and, a(x0, x), f(reqdeg, x, reqdeg1))))(-1.0)],
   current = {1} [l(x, f(reqdeg, x, reqdeg1))(-1.0)]
INFO: Trying to learn '(NP\NP)/NP' [requiring]: expected = {1} [l(x1
   , l(x0, l(x, g(and, a(x0, x), a(x1, x)))))(-1.0)], current = {1}
   [l(x, x)(-1.0)]
INFO: Trying to learn 'NP' [reqdeg1]: expected = {1} [l(x0, l(x, g(
   and, a(x0, x), f(reqdeg, x, reqdeg1))))(-2.0)], current = {1} [l(
   x, f(reqdeg, x, reqdeg1))(0.0)]
INFO: Trying to learn 'N' [reqdeg1]: expected = {1} [l(x0, l(x, g(
   and, a(x0, x), f(reqdeg, x, reqdeg1))))(-2.0)], current = {1} [l(
   x, f(reqdeg, x, reqdeg1))(0.0)]
INFO: Trying to learn 'N' [reqdeg1]: expected = {1} [l(x0, l(x, g(
   and, a(x0, x), f(reqdeg, x, reqdeg1))))(-2.0)], current = {1} [l(
   x, f(reqdeg, x, reqdeg1))(0.0)]
INFO: Trying to learn 'S' [Are there any jobs in loc1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)],
   current = {2} [l(x, f(answer, a(f(loc, l(x, f(job, x)), loc1), x)
   ))(-4.0), f(loc, l(x, f(job, x)), loc1)(-4.0)]
INFO: Trying to learn 'S' [there any jobs in loc1]: expected = {2} [
   l(x, g(and, f(job, x), f(loc, x, loc1)))(-1.0), l(x, f(answer, g(
   and, f(job, x), f(loc, x, loc1))))(-1.0)], current = {1} [f(loc,
   l(x, f(job, x)), loc1)(-3.0)]
INFO: Trying to learn 'S' [any jobs in loc1]: expected = {2} [l(x, g
   (and, f(job, x), f(loc, x, loc1)))(-2.0), l(x, f(answer, g(and, f
   (job, x), f(loc, x, loc1))))(-2.0)], current = {1} [f(loc, l(x, f
   (job, x)), loc1)(-2.0)]
INFO: Trying to learn 'S\NP' [in loc1]: expected = {2} [l(x0, l(x, g
   (and, a(x0, x), f(loc, x, loc1))))(-3.0), l(x0, l(x, f(answer, g(
   and, a(x0, x), f(loc, x, loc1)))))(-3.0)], current = {1} [l(x, f(
   loc, x, loc1))(-1.0)]
INFO: Trying to learn '(S\NP)/NP' [in]: expected = {2} [l(x1, l(x0,
   l(x, g(and, a(x0, x), a(x1, x)))))(-3.0), l(x1, l(x0, l(x, f(
   answer, g(and, a(x0, x), a(x1, x))))))(-3.0)], current = {1} [l(x
   , x)(-1.0)]
INFO: Trying to learn 'NP' [loc1]: expected = {2} [l(x0, l(x, g(and,
    a(x0, x), f(loc, x, loc1))))(-4.0), l(x0, l(x, f(answer, g(and,
   a(x0, x), f(loc, x, loc1)))))(-4.0)], current = {1} [l(x, f(loc,
   x, loc1))(0.0)]
INFO: Trying to learn 'N' [loc1]: expected = {1} [l(x0, l(x, g(and,
```

```
   a(x0, x), f(loc, x, loc1))))(-4.0)], current = {1} [l(x, f(loc, x
   , loc1))(0.0)]
INFO: Trying to learn 'N' [loc1]: expected = {2} [l(x0, l(x, g(and,
   a(x0, x), f(loc, x, loc1))))(-4.0), l(x0, l(x, f(answer, g(and, a
   (x0, x), f(loc, x, loc1)))))(-4.0)], current = {1} [l(x, f(loc, x
   , loc1))(0.0)]
INFO: Trying to learn 'N' [loc1]: expected = {2} [l(x0, l(x, g(and,
   a(x0, x), f(loc, x, loc1))))(-4.0), l(x0, l(x, f(answer, g(and, a
   (x0, x), f(loc, x, loc1)))))(-4.0)], current = {1} [l(x, f(loc, x
   , loc1))(0.0)]
INFO: Trying to learn 'S' [Are there any jobs specializing in area1
   with company1]: expected = {1} [l(x, f(answer, g(and, f(job, x),
   f(area, x, area1), f(company, x, company1))))(0.0)], current =
   {0} []
INFO: Trying to learn 'S' [there any jobs specializing in area1 with
    company1]: expected = {2} [l(x, g(and, f(job, x), f(area, x,
   area1), f(company, x, company1)))(-1.0), l(x, f(answer, g(and, f(
   job, x), f(area, x, area1), f(company, x, company1))))(-1.0)],
   current = {0} []
INFO: Trying to learn 'S' [any jobs specializing in area1 with
   company1]: expected = {2} [l(x, g(and, f(job, x), f(area, x,
   area1), f(company, x, company1)))(-2.0), l(x, f(answer, g(and, f(
   job, x), f(area, x, area1), f(company, x, company1))))(-2.0)],
   current = {0} []
INFO: Trying to learn 'S\NP' [specializing in area1 with company1]:
   expected = {2} [l(x0, l(x, g(and, a(x0, x), f(area, x, area1), f(
   company, x, company1))))(-3.0), l(x0, l(x, f(answer, g(and, a(x0,
    x), f(area, x, area1), f(company, x, company1)))))(-3.0)],
   current = {0} []
INFO: Trying to learn '(S\NP)\(S\NP)' [in area1 with company1]:
   expected = {2} [l(x1, a(x1, l(x0, l(x, g(and, a(x0, x), f(area, x
   , area1), f(company, x, company1))))))(-4.0), l(x1, a(x1, l(x0, l
   (x, f(answer, g(and, a(x0, x), f(area, x, area1), f(company, x,
   company1)))))))(-4.0)], current = {1} [f(company, l(x, f(area, x,
    area1)), company1)(-2.0)]
INFO: Trying to learn 'NP' [area1 with company1]: expected = {2} [l(
   x1, a(x1, l(x0, l(x, g(and, a(x0, x), f(area, x, area1), f(
   company, x, company1))))))(-5.0), l(x1, a(x1, l(x0, l(x, f(answer
   , g(and, a(x0, x), f(area, x, area1), f(company, x, company1)))))
   ))(-5.0)], current = {1} [f(company, l(x, f(area, x, area1)),
   company1)(-1.0)]
INFO: Trying to learn 'NP\NP' [with company1]: expected = {2} [l(x2,
    l(x1, a(x1, l(x0, l(x, g(and, a(x0, x), a(x2, x), f(company, x,
   company1)))))))(-5.0), l(x2, l(x1, a(x1, l(x0, l(x, f(answer, g(
   and, a(x0, x), a(x2, x), f(company, x, company1)))))))(-5.0)],
   current = {1} [l(x, f(company, x, company1))(-1.0)]
INFO: Trying to learn '(NP\NP)/NP' [with]: expected = {2} [l(x3, l(
   x2, l(x1, a(x1, l(x0, l(x, g(and, a(x0, x), a(x2, x), a(x3, x))))
   ))))(-5.0), l(x3, l(x2, l(x1, a(x1, l(x0, l(x, f(answer, g(and, a
   (x0, x), a(x2, x), a(x3, x)))))))))(-5.0)], current = {1} [l(x, x
   )(-1.0)]
INFO: Trying to learn 'NP' [company1]: expected = {2} [l(x2, l(x1, a
   (x1, l(x0, l(x, g(and, a(x0, x), a(x2, x), f(company, x, company1
   ))))))))(-6.0), l(x2, l(x1, a(x1, l(x0, l(x, f(answer, g(and, a(x0
   , x), a(x2, x), f(company, x, company1)))))))))(-6.0)], current =
   {1} [l(x, f(company, x, company1))(0.0)]
INFO: Trying to learn 'N' [company1]: expected = {1} [l(x2, l(x1, a(
```

```
   x1, l(x0, l(x, g(and, a(x0, x), a(x2, x), f(company, x, company1)
   ))))))(-6.0)], current = {1} [l(x, f(company, x, company1))(0.0)]
INFO: Trying to learn 'N' [company1]: expected = {2} [l(x2, l(x1, a(
   x1, l(x0, l(x, g(and, a(x0, x), a(x2, x), f(company, x, company1)
   ))))))(-6.0), l(x2, l(x1, a(x1, l(x0, l(x, f(answer, g(and, a(x0,
    x), a(x2, x), f(company, x, company1)))))))(-6.0)], current =
   {1} [l(x, f(company, x, company1))(0.0)]
INFO: Trying to learn 'N' [company1]: expected = {2} [l(x2, l(x1, a(
   x1, l(x0, l(x, g(and, a(x0, x), a(x2, x), f(company, x, company1)
   ))))))(-6.0), l(x2, l(x1, a(x1, l(x0, l(x, f(answer, g(and, a(x0,
    x), a(x2, x), f(company, x, company1)))))))(-6.0)], current =
   {1} [l(x, f(company, x, company1))(0.0)]
INFO: Learned 'reqdeg1 : N : l(x0, l(x, g(and, a(x0, x), f(reqdeg, x
   , reqdeg1))))' by inverse
INFO: Learned 'company1 : N : l(x2, l(x1, a(x1, l(x0, l(x, g(and, a(
   x0, x), a(x2, x), f(company, x, company1)))))))' by inverse
INFO: Learned 'company1 : N : l(x2, l(x1, a(x1, l(x0, l(x, f(answer,
    g(and, a(x0, x), a(x2, x), f(company, x, company1)))))))' by
   inverse
INFO: Learned 'in : (S\NP)/NP : l(x1, l(x0, l(x, f(answer, g(and, a(
   x0, x), a(x1, x))))))' by inverse
INFO: Learned 'requiring : (NP\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0
   , x), a(x1, x)))))' by inverse
INFO: Learned 'with : (NP\NP)/NP : l(x3, l(x2, l(x1, a(x1, l(x0, l(x
   , g(and, a(x0, x), a(x2, x), a(x3, x)))))))' by inverse
INFO: Learned 'with : (NP\NP)/NP : l(x3, l(x2, l(x1, a(x1, l(x0, l(x
   , f(answer, g(and, a(x0, x), a(x2, x), a(x3, x)))))))))' by
   inverse
INFO: Learned 'loc1 : N : l(x0, l(x, g(and, a(x0, x), f(loc, x, loc1
   ))))' by inverse
INFO: Learned 'loc1 : N : l(x0, l(x, f(answer, g(and, a(x0, x), f(
   loc, x, loc1)))))' by inverse
INFO: Learned 'requiring : (NP\NP)/NP : l(x3, l(x2, l(x1, a(x1, l(x0
   , l(x, g(and, a(x0, x), a(x2, x), a(x3, x)))))))' by
   generalization
INFO: Learned 'requiring : (NP\NP)/NP : l(x3, l(x2, l(x1, a(x1, l(x0
   , l(x, f(answer, g(and, a(x0, x), a(x2, x), a(x3, x)))))))))' by
   generalization
INFO: Learned 'with : (NP\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0, x),
    a(x1, x)))))' by generalization
INFO: We learned 12 new lexicon(s):
```

In this iteration, using the new generalized meanings, we can learn all of the remaining sentences. We actually have all of the needed meanings for one sentence in those remaining, so we only need to verify; for the rest, we must use inverse learning to derive the missing meanings.

Once we learn one word by inversion, we use this meaning to generalize new meanings for words on the waiting list (subsection 6.4.2). Overall, we learn 12 new meanings in this iteration.

Iteration 5:

```
SEVERE: ============================ Iteration 5
   ============================
INFO: Trying to learn 'S' [List jobs requiring reqdeg1 using
   language1]: expected = {1} [l(x, f(answer, g(and, f(job, x), f(
   reqdeg, x, reqdeg1), f(language, x, language1))))(0.0)], current
```

```
   = {88} ...
INFO: Trying to learn 'S' [Are there any jobs in loc1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)],
   current = {20} ...
INFO: Trying to learn 'S' [Are there any jobs specializing in area1
   with company1]: expected = {1} [l(x, f(answer, g(and, f(job, x),
   f(area, x, area1), f(company, x, company1))))(0.0)], current =
   {16} ...
INFO: We learned 0 new lexicon(s):
```

Similar to iteration 2, here we verify learning of the remaining sentences.

Iteration 6:

```
SEVERE: ========================== Iteration 6
   ==========================
INFO: We fully learned 7 of 7 sentences: [0, 1, 2, 3, 4, 5, 6]
INFO: Incompleted learning sentences: []
INFO: Lexicon Learning costs: 00h:00m:01s:238ms
```

After verifying that all sentences have been learned, we cease the lexicon learning process and begin the parameter estimation.

Parameter Estimation:

```
INFO: ******Parameter estimation ...
INFO: Current Parser:SPARSE
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Parse all the sentences again to get all possible parse trees
INFO: Updating the feature vector table
INFO: Initial theta = [0.0, 1.3862943611198906, 1.3862943611198906,
   2.0794415416798357, 1.3862943611198906, 0.6931471805599453,
   0.6931471805599453, 0.6931471805599453, 0.0, 1.0986122886681098,
   -1.0, 2.8903717578961645, -1.0, 1.0986122886681098,
   0.6931471805599453, 0.0, 0.0, -1.0, 0.0, -1.0, 1.791759469228055,
    0.6931471805599453, 0.0, 0.0, 1.0986122886681098,
   1.3862943611198906, 1.0986122886681098, 0.0, 0.0, -1.0, -1.0,
   2.0794415416798357, 1.791759469228055, 1.0986122886681098,
   1.3862943611198906, 1.3862943611198906]
INFO: Number of features = 36
INFO: Using simple parameter estimation.
INFO: Parameter Estimation costs: 00h:00m:01s:285ms
INFO: ******Evaluation on training set ...
INFO: Generalizing using = [using : (S\NP)/NP : l(x1, l(x0, l(x, f(
   answer, g(and, a(x0, x), a(x1, x)))))), using : (S\NP)/NP : l(x,
   x)]
INFO: Generalizing requiring = [requiring : (S\NP)/NP : l(x1, l(x0,
   l(x, f(answer, g(and, a(x0, x), a(x1, x)))))), requiring : (S\NP)
   /NP : l(x, x)]
```

ID	Sentence	Meaning
1	List jobs in loc1	λ x.answer(x,job(x) ∧ loc(x,loc1))
2	List jobs in area1	λ x.answer(x,job(x) ∧ area(x,area1))
3	Show jobs using language1	λ x.answer(x,job(x) ∧ language(x,language1))
4	List jobs requiring reqdeg1	λ x.answer(x,job(x) ∧ req_deg(x,reqdeg1))
5	List jobs requiring reqdeg1 using language1	λ x.answer(x,job(x) ∧ req_deg(x,reqdeg1) ∧ language(x,language1))
6	Are there any jobs in loc1	λ x.answer(x,job(x) ∧ loc(x,loc1))
7	Are there any jobs specializing in area1 with company1	λ x.answer(x,job(x) ∧ area(x,area1) ∧ company(x,company1))
8	Are there any jobs requiring a reqdeg1 for company1 in loc1	λ x.answer(x,job(x) ∧ req_deg(x,reqdeg1) ∧ company(x,company1) ∧ loc(x,loc1))
9	Are there any jobs in loc1 requiring at least a reqdeg1 and knowing language1	λ x.answer(x,job(x) ∧ loc(x,loc1) ∧ req_deg(x,reqdeg1) ∧ language(x,language1))

Table D.3: Sentences and Their Meanings in the JobsCompact9 corpus.

```
INFO: Evaluation costs: 00h:00m:01s:199ms
INFO: ******Correct Parse :7 out of 7
INFO: Total training costs: 00h:00m:03s:734ms
```

The parameter estimation process learns the weights for each meaning in the lexicon. After that, it evaluates the training set and observes that we can correctly translate 7 out of 7 sentences.

Shown in Figure D.10 to Figure D.16 are the meanings of each word in the sentences and how they are combined for the correct meanings of the entire sentences.

D.2 Training on JobsCompact9 Corpus

Now we add 2 more sentences to JobCompact7 to create the JobCompact9 corpus; these two sentences are more complex than the rest. The new training corpus is shown in Table D.3. The CCG parse trees of the two new sentences are shown in Figure D.17 and D.18.

Using the same initial dictionary as JobsCompact7, we cannot learn the two new sentences, since they have new word "requiring" with different CCG category than the word "requiring" in other sentences. Because this word has different categories, its meaning cannot be generalized from the previous. Moreover, the two sentences also have other unknown words; the inverse algorithm cannot be used to learn the meaning of "requiring." The solution is either making sure that the two sentences miss only the meaning of "requiring" or providing the meaning of "requiring" and allow the algorithm to learn many other words in the two sentences.

We choose to provide the meaning of "requiring" λ p. λ q. λ x.q @ x ∧ p @ x, which

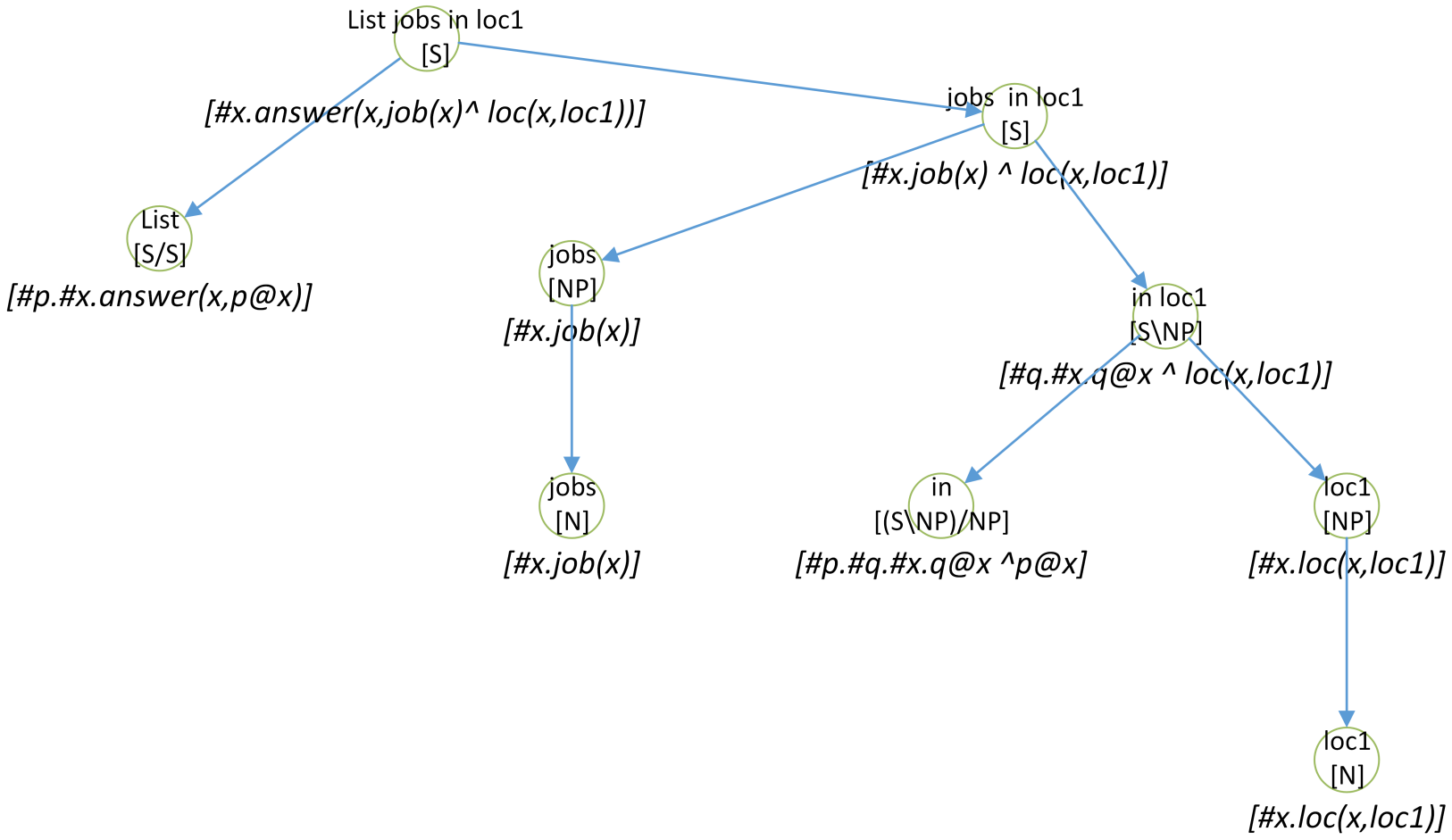

Figure D.10: Meanings used to get the correct meaning of sentence 1

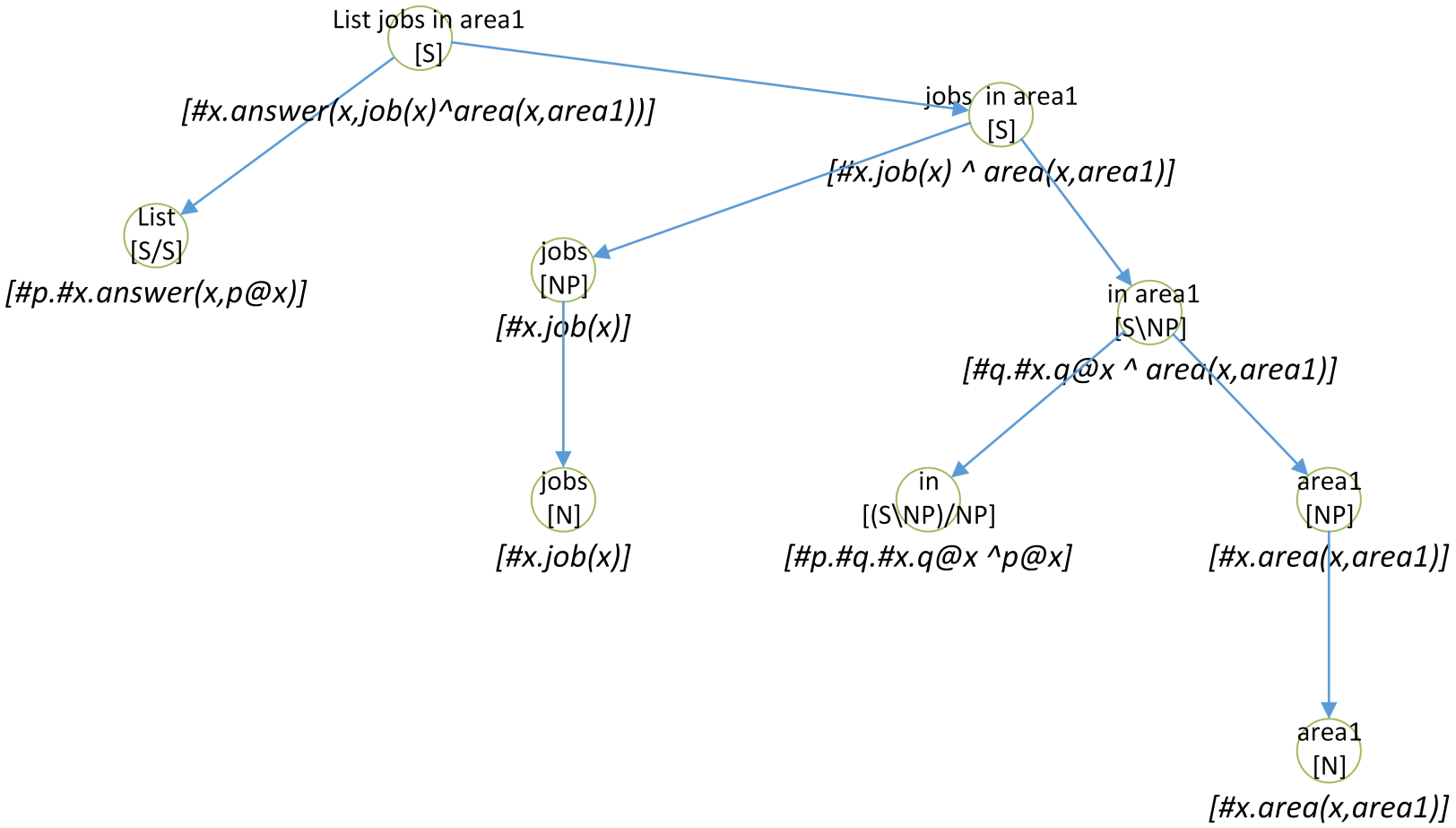

Figure D.11: Meanings used to get the correct meaning of sentence 2

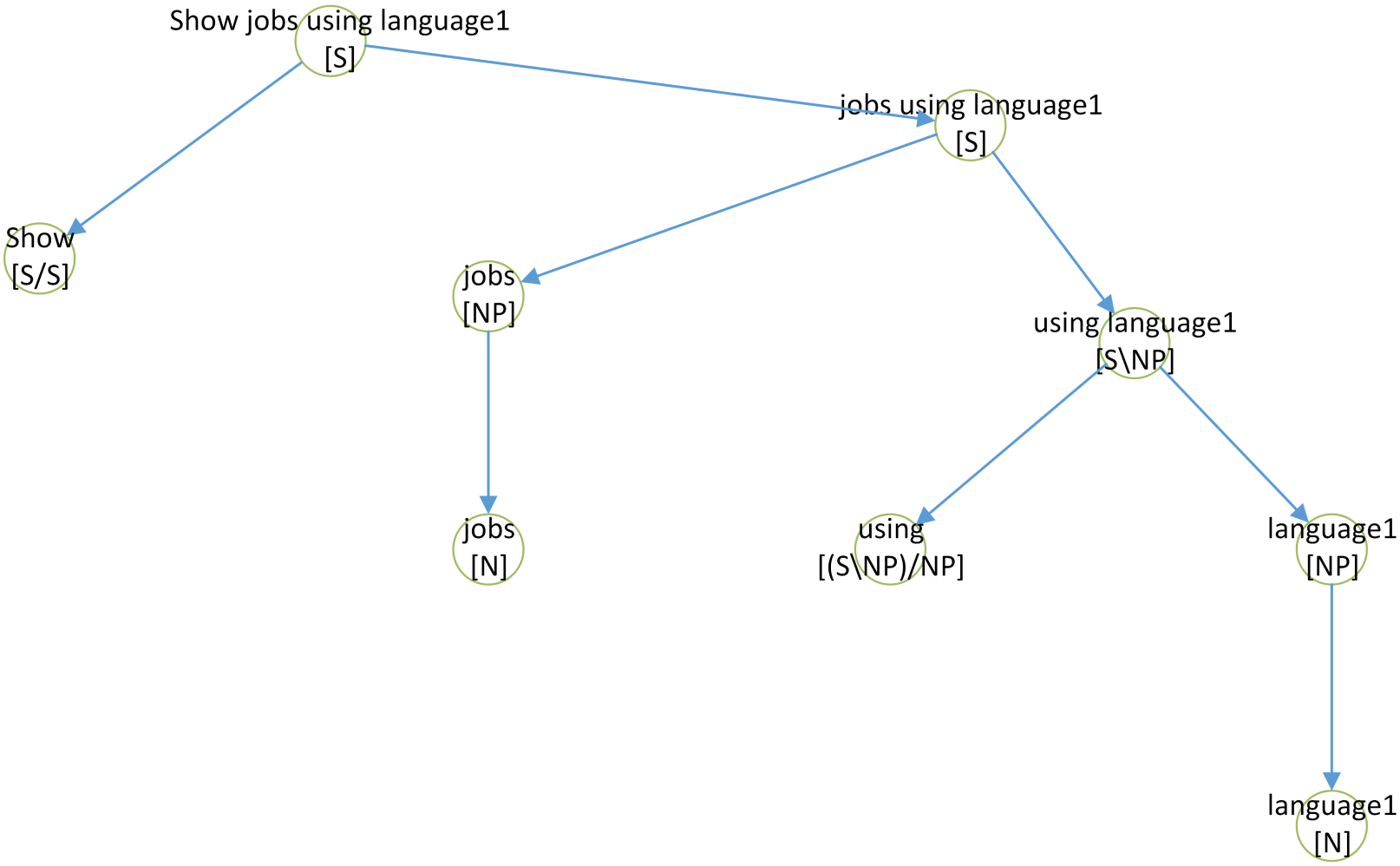

Figure D.12: Meanings used to get the correct meaning of sentence 3

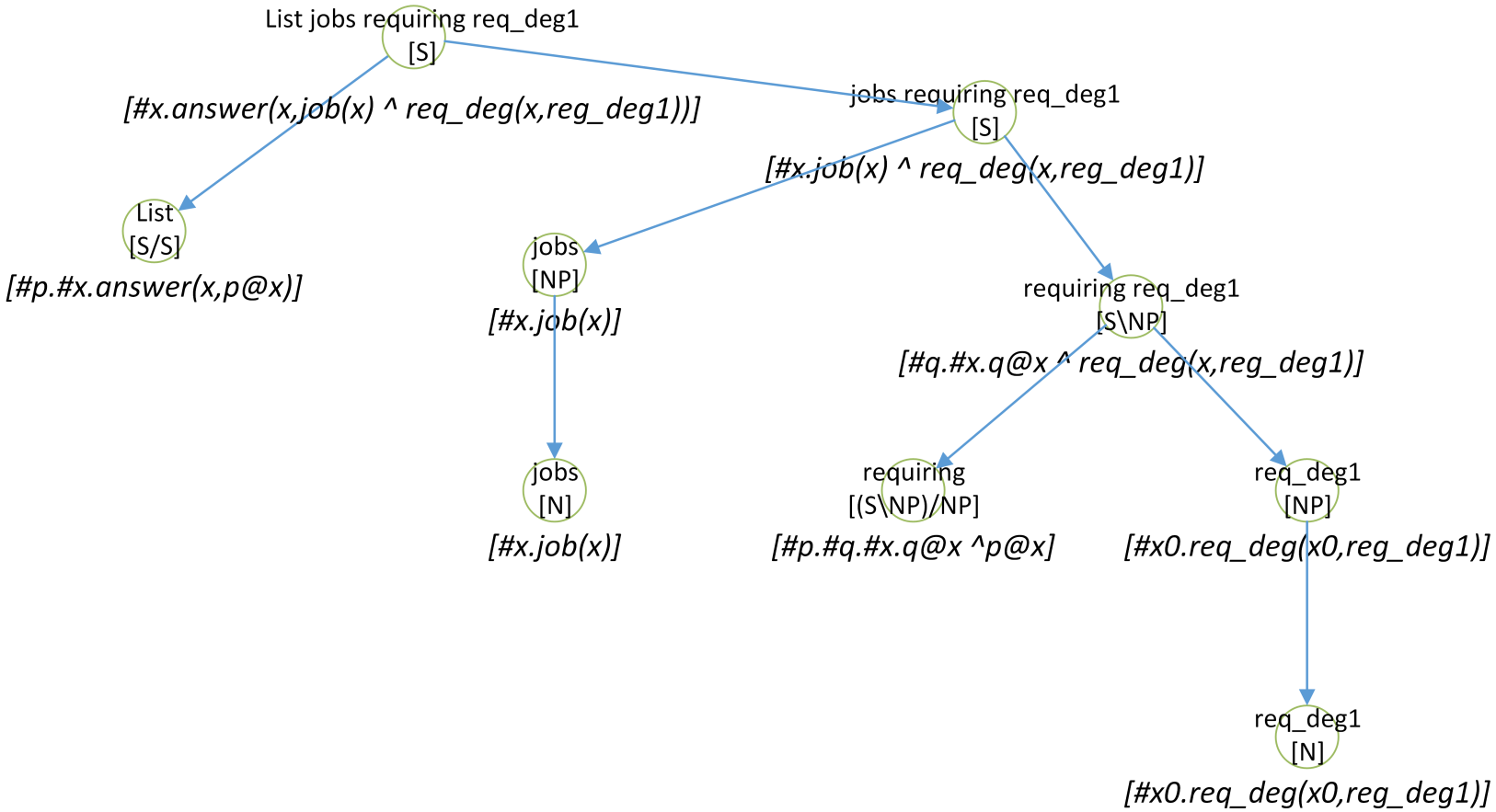

Figure D.13: Meanings used to get the correct meaning of sentence 4

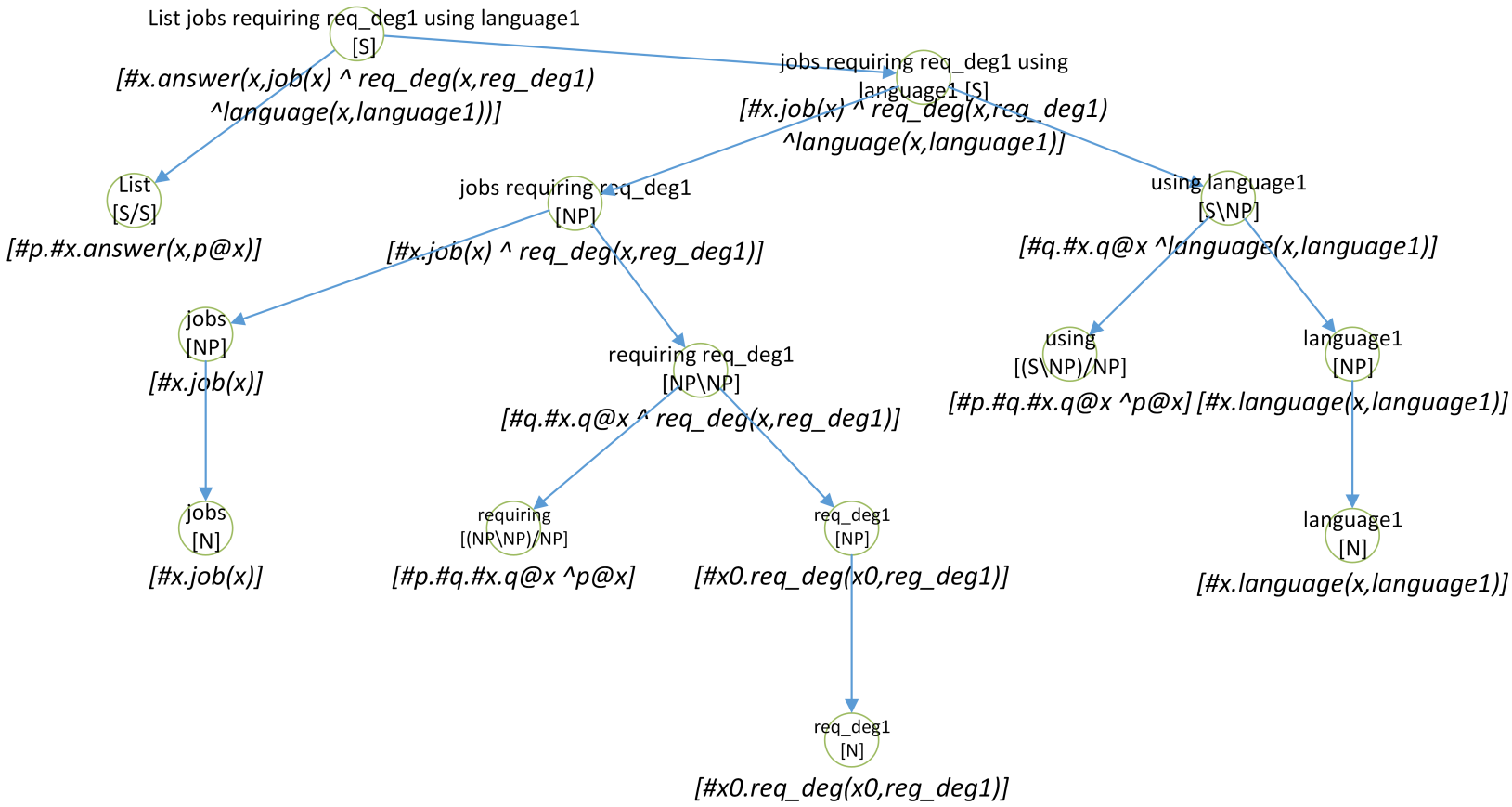

Figure D.14: Meanings used to get the correct meaning of sentence 5

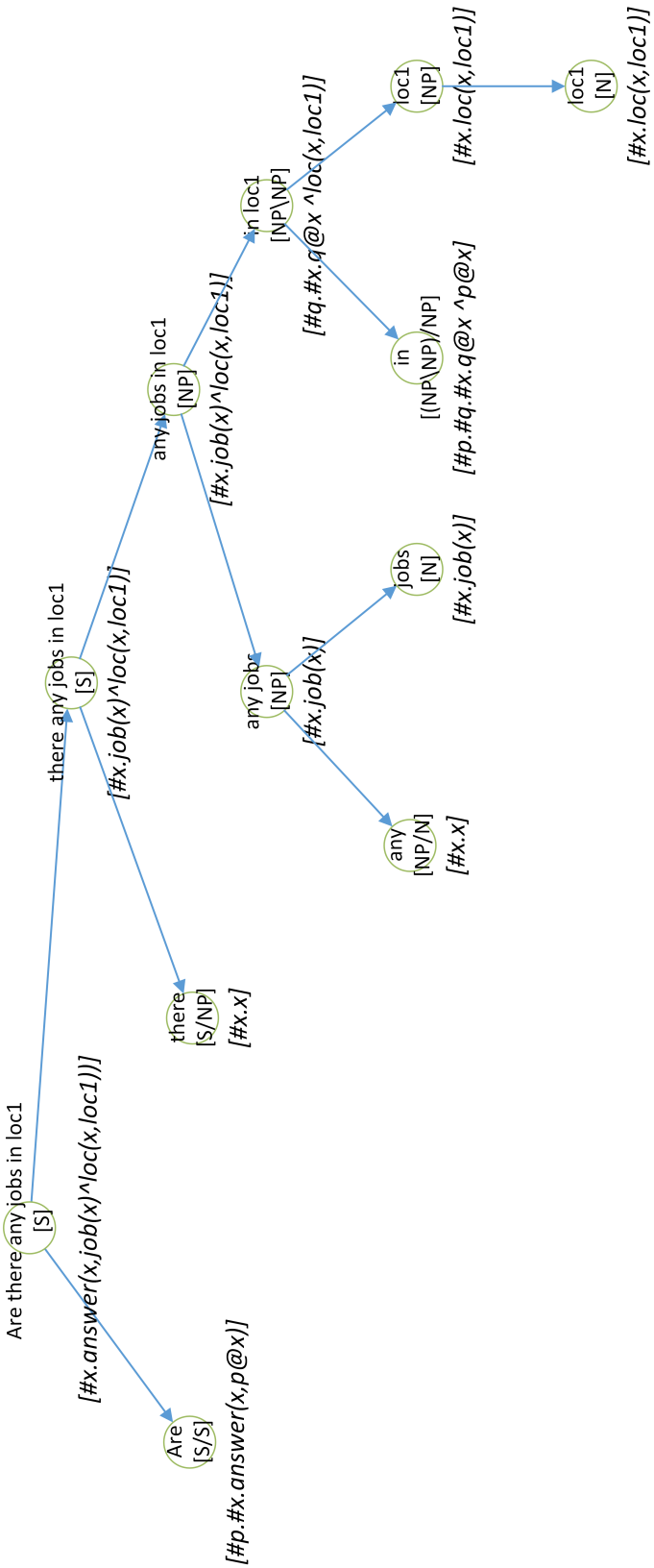

Figure D.15: Meanings used to get the correct meaning of sentence 6

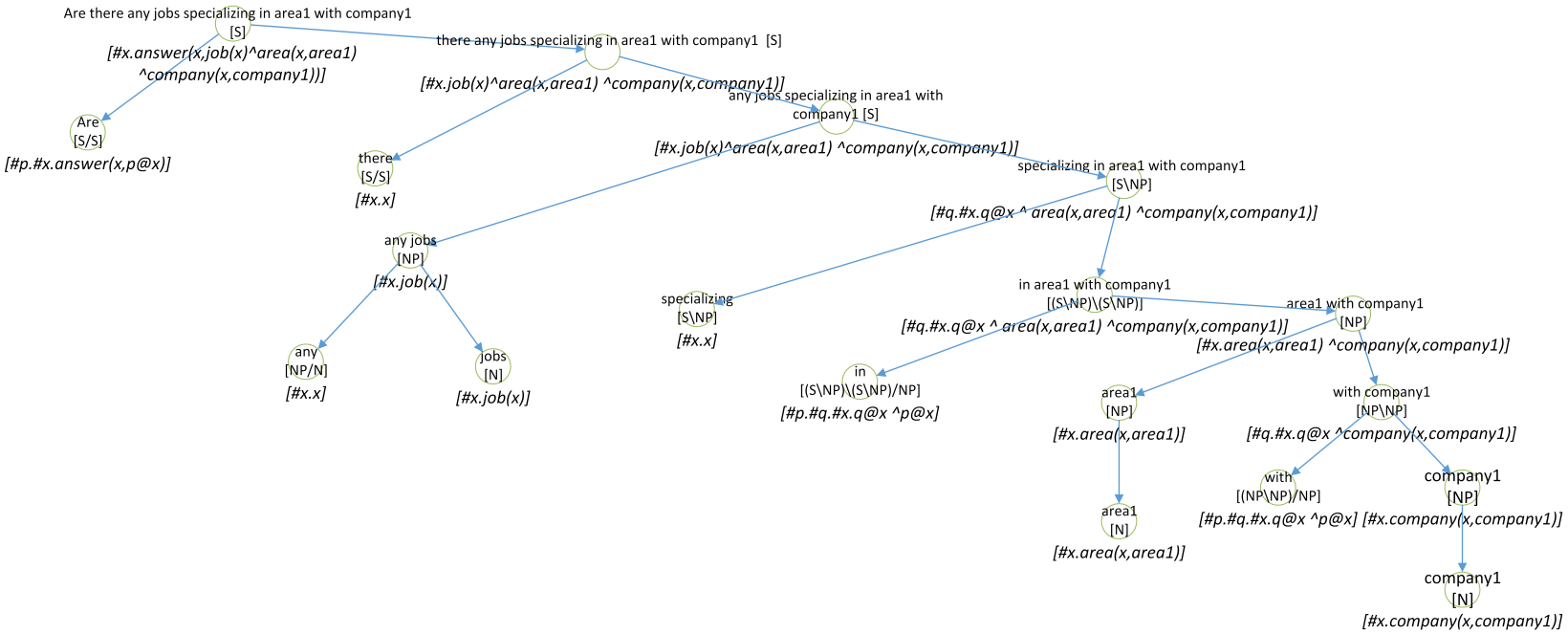

Figure D.16: Meanings used to get the correct meaning of sentence 7

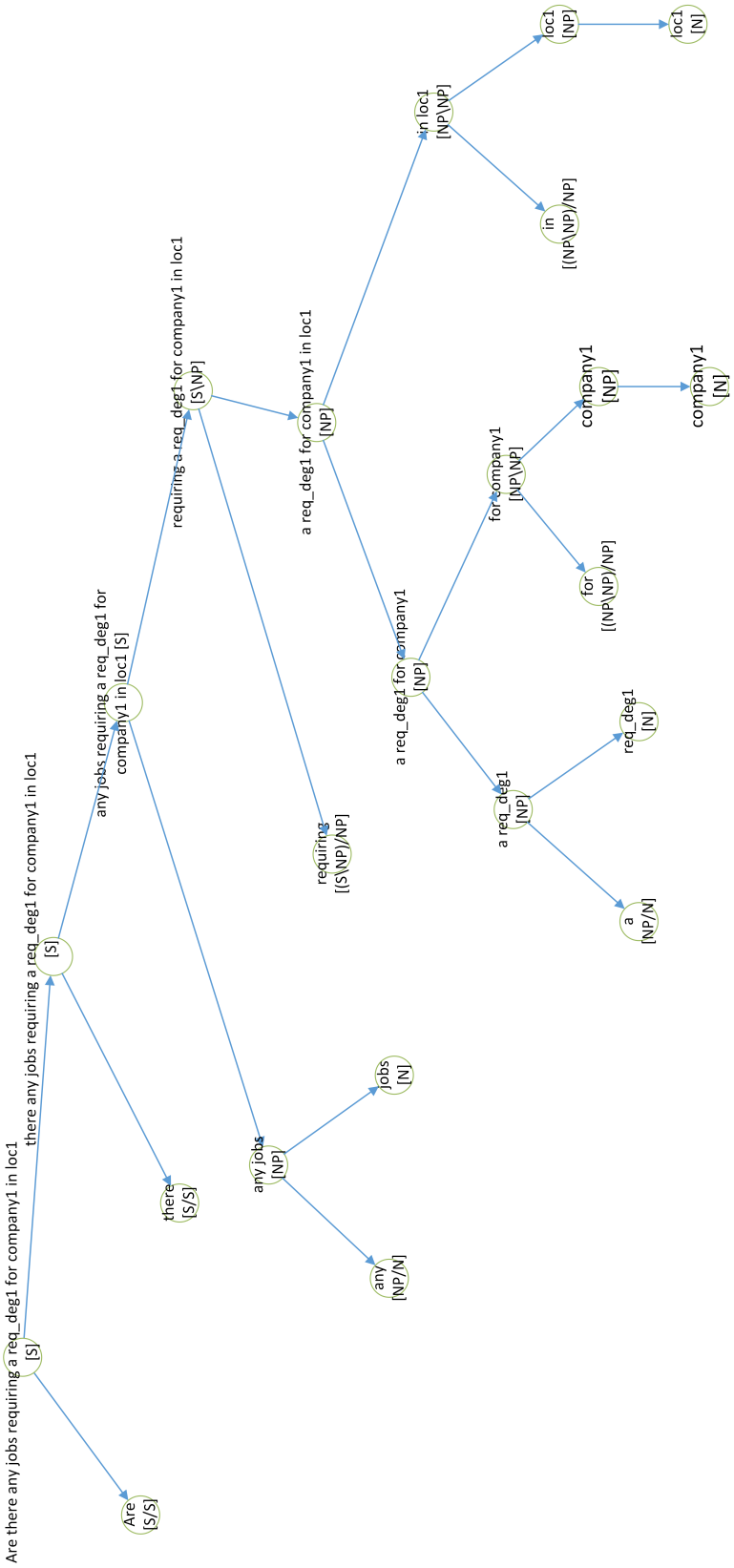

Figure D.17: CCG parse tree of sentence 8

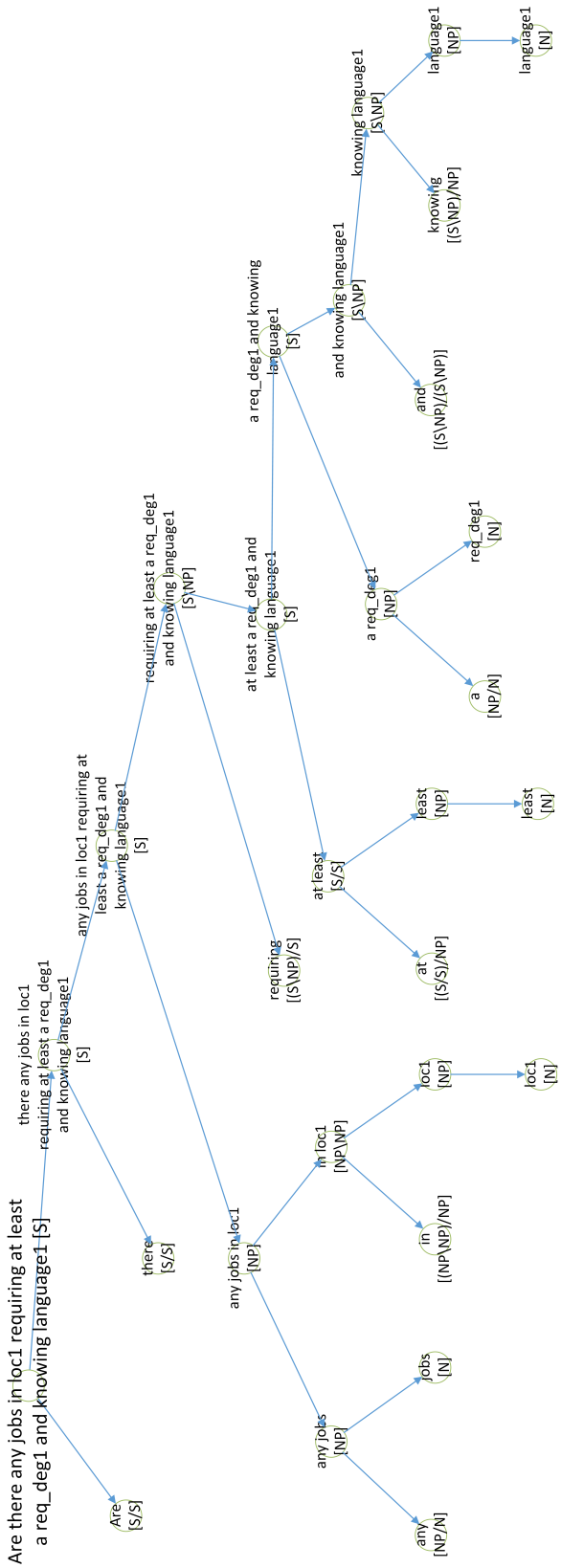

Figure D.18: CCG parse tree of sentence 9

Word	CCG	Meaning
List	S/S	λ p. λ x.answer(x,p @ x)
requiring	(NP $\setminus$ NP)/NP	λ p. λ q. λ x.q @ x $\wedge$ p @ x
jobs	N	λ x.job(x)
area1	N	λ x.area(x,area1)
company1	N	λ x.company(x,company1)
language1	N	λ x.language(x,language1)
loc1	N	λ x.loc(x,loc1)
reqdeg1	N	λ x.req_deg(x,reqdeg1)
least	N	λ x.x

Table D.4: Initial Dictionary for JobsCompact9 Corpus

is the same as the meanings of "in" and "for" in previous case. The new initial dictionary thus is shown in Table D.4.

Learning on the new corpus is a little bit different from the previous.

```
SEVERE: ========================== Iteration 1
   ==========================
INFO: Trying to learn 'S' [List jobs in loc1]: expected = {1} [l(x,
   f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)], current =
   {0} []
INFO: Trying to learn 'S' [jobs in loc1]: expected = {1} [l(x, g(and
   , f(job, x), f(loc, x, loc1)))(0.0)], current = {0} []
INFO: Trying to learn 'S\NP' [in loc1]: expected = {1} [l(x0, l(x, g
   (and, a(x0, x), f(loc, x, loc1))))(0.0)], current = {0} []
INFO: Trying to learn '(S\NP)/NP' [in]: expected = {1} [l(x1, l(x0,
   l(x, g(and, a(x0, x), a(x1, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [List jobs in area1]: expected = {1} [l(x,
    f(answer, g(and, f(job, x), f(area, x, area1))))(0.0)], current
   = {0} []
INFO: Trying to learn 'S' [jobs in area1]: expected = {1} [l(x, g(
   and, f(job, x), f(area, x, area1)))(0.0)], current = {0} []
INFO: Trying to learn 'S\NP' [in area1]: expected = {1} [l(x0, l(x,
   g(and, a(x0, x), f(area, x, area1))))(0.0)], current = {0} []
INFO: Trying to learn '(S\NP)/NP' [in]: expected = {1} [l(x1, l(x0,
   l(x, g(and, a(x0, x), a(x1, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [Show jobs using language1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(language, x, language1)))
   )(0.0)], current = {0} []
INFO: Trying to learn 'S' [List jobs requiring reqdeg1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(reqdeg, x, reqdeg1))))
   (0.0)], current = {0} []
INFO: Trying to learn 'S' [jobs requiring reqdeg1]: expected = {1} [
   l(x, g(and, f(job, x), f(reqdeg, x, reqdeg1)))(0.0)], current =
   {0} []
INFO: Trying to learn 'S\NP' [requiring reqdeg1]: expected = {1} [l(
   x0, l(x, g(and, a(x0, x), f(reqdeg, x, reqdeg1))))(0.0)], current
    = {0} []
INFO: Trying to learn '(S\NP)/NP' [requiring]: expected = {1} [l(x1,
    l(x0, l(x, g(and, a(x0, x), a(x1, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [List jobs requiring reqdeg1 using
   language1]: expected = {1} [l(x, f(answer, g(and, f(job, x), f(
```

```
   reqdeg, x, reqdeg1), f(language, x, language1))))(0.0)], current
   = {0} []
INFO: Trying to learn 'S' [jobs requiring reqdeg1 using language1]:
   expected = {1} [l(x, g(and, f(job, x), f(reqdeg, x, reqdeg1), f(
   language, x, language1)))(0.0)], current = {0} []
INFO: Trying to learn 'S\NP' [using language1]: expected = {1} [l(x0
   , l(x, g(and, f(language, x, language1), a(x0, x))))(0.0)],
   current = {0} []
INFO: Trying to learn '(S\NP)/NP' [using]: expected = {1} [l(x1, l(
   x0, l(x, g(and, a(x1, x), a(x0, x)))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [Are there any jobs in loc1]: expected =
   {1} [l(x, f(answer, g(and, f(job, x), f(loc, x, loc1))))(0.0)],
   current = {0} []
INFO: Trying to learn 'S' [Are there any jobs specializing in area1
   with company1]: expected = {1} [l(x, f(answer, g(and, f(job, x),
   f(area, x, area1), f(company, x, company1))))(0.0)], current =
   {0} []
INFO: Trying to learn 'S' [Are there any jobs requiring a reqdeg1
   for company1 in loc1]: expected = {1} [l(x, f(answer, g(and, f(
   job, x), f(reqdeg, x, reqdeg1), f(company, x, company1), f(loc, x
   , loc1))))(0.0)], current = {0} []
INFO: Trying to learn 'S' [Are there any jobs in loc1 requiring at
   least a reqdeg1 and knowing language1]: expected = {1} [l(x, f(
   answer, g(and, f(job, x), f(loc, x, loc1), f(reqdeg, x, reqdeg1),
    f(language, x, language1))))(0.0)], current = {0} []
INFO: Learned 'in : (S\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0, x), a(
   x1, x)))))' by inverse
INFO: Learned 'requiring : (S\NP)/NP : l(x1, l(x0, l(x, g(and, a(x0,
    x), a(x1, x)))))' by inverse
INFO: Learned 'using : (S\NP)/NP : l(x1, l(x0, l(x, g(and, a(x1, x),
    a(x0, x)))))' by inverse
INFO: We learned 3 new lexicon(s):
```

Knowing more allows us to fully learn 4 sentences (previously 3) in iteration 1. After verification in iteration 2 and generalization in iteration 3, we learn the remaining sentences in iteration 4. In iteration 5, 5 remaining sentences are found be finished. In iteration 6, all sentences are finished and parameter estimation is called.

Complete parse tree with meanings of the two new sentences are shown in Figure D.19 and D.20

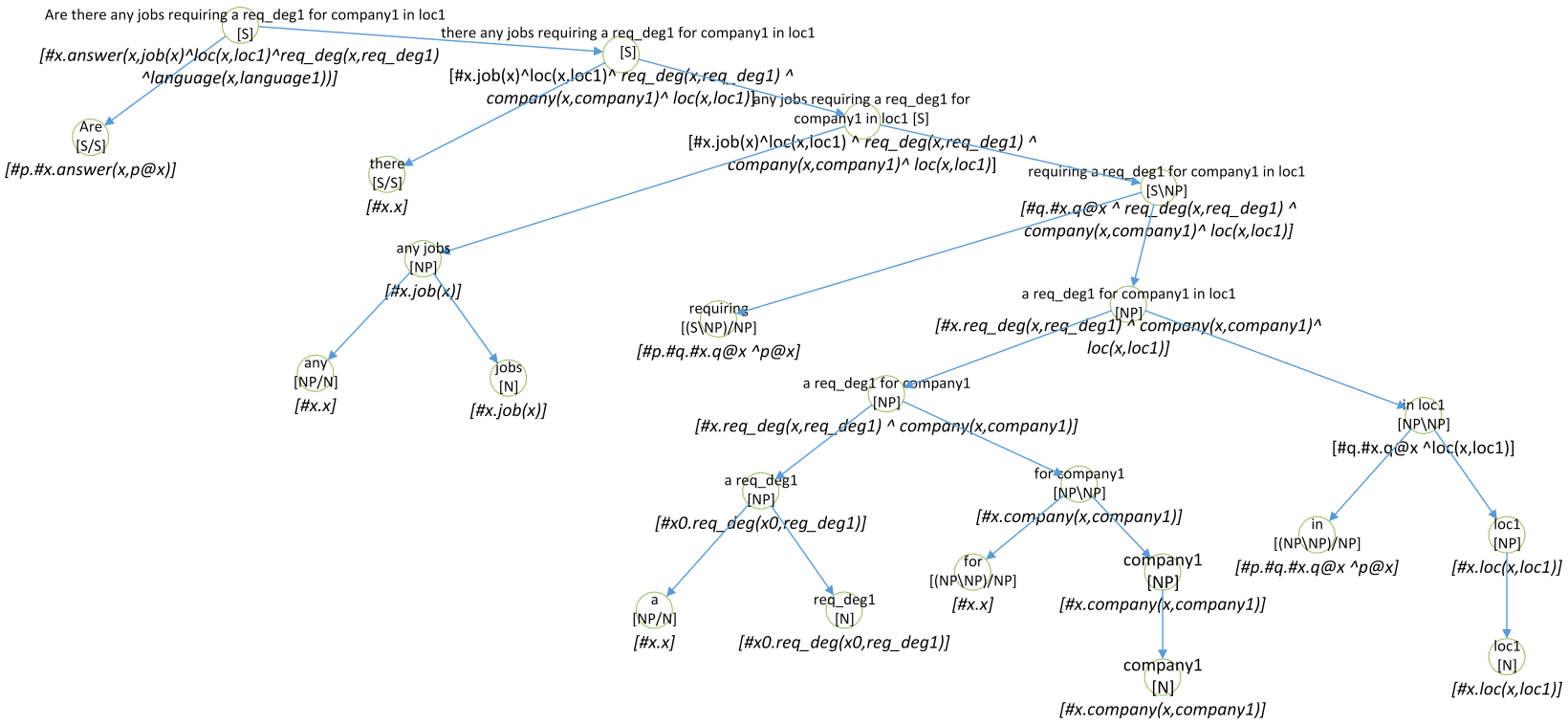

Figure D.19: Meanings used to get the correct meaning of sentence 1

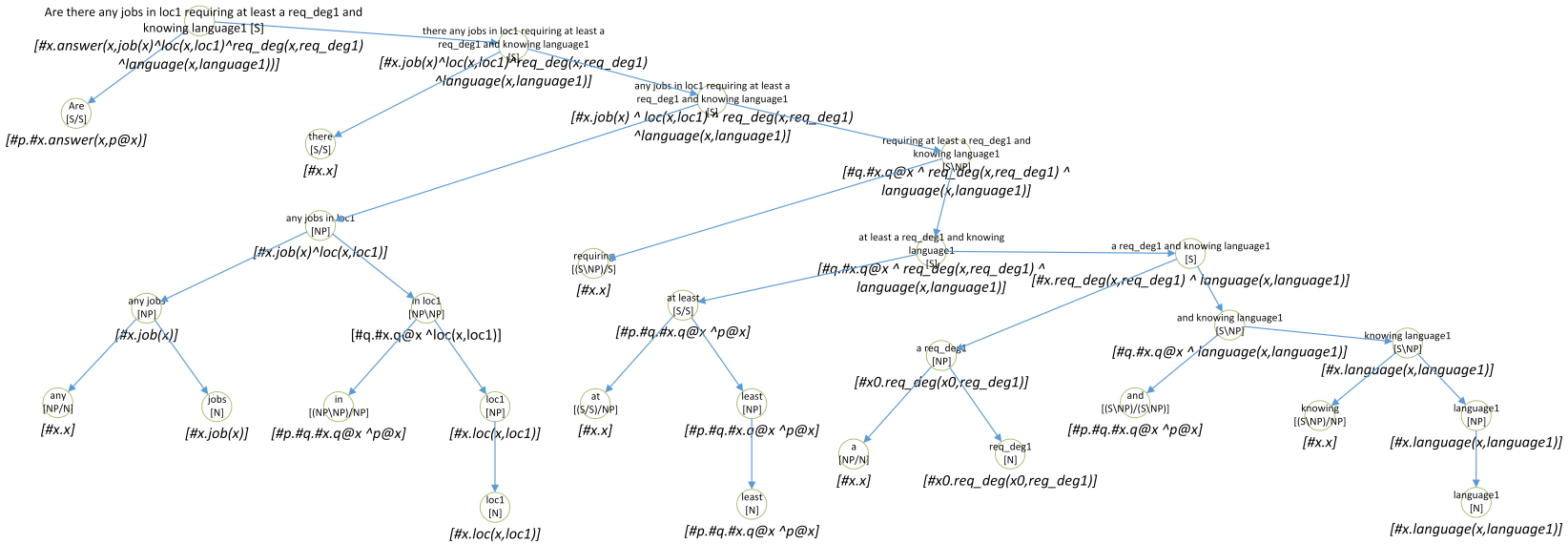

Figure D.20: Meanings used to get the correct meaning of sentence 1