1 Glue Semantics

Files associated with this chapter (to be placed in the 'ntlk_lite/semantics' directory):

- http://www.iwu.edu/~dgarrett/programming/ntlk/glue/glue.py
- http://www.iwu.edu/~dgarrett/programming/ntlk/glue/linearlogic.py
- http://www.iwu.edu/~dgarrett/programming/ntlk/glue/grammar.cfg
- http://www.iwu.edu/~dgarrett/programming/ntlk/glue/glue.cfg

1.1 Introduction

This chapter expands on the ideas about Semantic Interpretation presented in Chapter 11. The **Principle of Compositionality** states that the meaning of a whole is a function of the meanings of the parts and of the way they are syntactically combined. Chapter 11 described an algorithm to perform this composition. However, as it was noted in the chapter, there are some limitations that quickly arise. For example, note (43), (44a), and (44b) from Chapter 11 repeated here as (1)–(3) respectively.

(1) Every girl chases a dog.

(2) all x.((girl x) implies some y. ((dog y) and (chase x y)))

(3) some y.((dog y) and all x. ((girl x) implies (chase x y)))

In this example, (1) should give rise to both readings (2) and (3). However, the procedure described in Chapter 11 only generates (2). The reason for this discrepancy is that semantic interpretation in Chapter 11 is too closely tied to syntax. In sentence (2), the word "every" out-scopes the word "a". This is a result of the sentence's word order: "every" comes before "a". In the semantic composition, "every" combines with "boy" and "a" with "girl". But since "a girl" is a sub-part of the VP "chases a girl", "a girl" is combined with "chases" before "every boy" is.

In contrast, the topic of this chapter, Glue Semantics, offers an elegant approach to determining scope based on the use of resource-sensitive logic as a means to "glue" the λ-calculus meaning terms together.

1.2 Linear Logic

The particular "glue" logic that we will use is **Implicational Linear Logic**. It is a subset of Propositional Logic (see Chapter 11.3). The logic is "implicational" because it's only operator is the implication. However, we will use the symbol '$\multimap$' for linear logic implication instead of the symbol '$\rightarrow$' which was used in the propositional and first order logics. Just as in propositional logic, the implication is used in the following way:

(4) $\mathrm{A}, (\mathrm{A} \multimap \mathrm{B}) \vdash \mathrm{B}$

Linear Logic is **resource-sensitive** because every premise in a resource-sensitive proof must be used <u>once and only once</u>. Therefore, unlike propositional or first order logics, we have the following rules:

(5) $\mathrm{A}, (\mathrm{A} \multimap \mathrm{B}) \nvdash \mathrm{A}, \mathrm{B}$

(6) $\mathrm{A}, \mathrm{A}, (\mathrm{A} \multimap \mathrm{B}) \nvdash \mathrm{B}$

The best way to think about these rules is consider the linear logic implicational statement as a function that **consumes** its antecedent and itself to **produce** its consequent. Example (4) is valid because A is applied to (A $\multimap$ B) which consumes both and produces B. On the other hand (5) is <u>invalid</u> because the premise A is consumed by the implication and therefore cannot appear in the conclusion. Finally, (6) is <u>invalid</u> because the premise A appears twice, but there is only one implication to consume it, and since every premise must be used exactly once, the second instance of A cannot be ignored.

1.2.1 Linear Logic in NLTK

All the tools needed to work in implicational linear logic are found in the file "linearlogic.py", which should be placed in the "semantics" folder. This module is modeled after the module logic.py. Strings can be parsed using the class Parser; ApplicationExpressions can be applied to other Expressions using the method applyto(). Note that applyto() will only succeed if the application is legal, otherwise it will raise an exception.

```
>>>from nltk_lite.semantics import linearlogic
>>>llp = linearlogic.Parser()
>>>p = llp.parse('p')
>>>print p
p
>>>p_q = llp.parse('(p -o q)')
>>>print p_q
(-o p q)
>>>print p_q.infixify()
(p -o q)
>>>q = p_q.applyto(p)
>>>q
ApplicationExpression('(-o p q)', 'p') []
>>>print q.simplify().infixify()
q
>>>p = p_q.applyto(q)
LinearLogicApplicationError: Attempting to apply (-o p q) to q
```

The module also binds variables correctly and stores those bindings for future applications. The when printing an Expression, the bindings are displayed as a list at the end.

```
>>>gf = llp.parse('(g -o f)')
>>>gGG = llp.parse('((g -o G) -o G)')
>>>f = gGG.applyto(gf)
>>>f
ApplicationExpression('(-o (-o g G) G)', '(-o g f)') [(G, f)]
>>>print f.simplify().infixify()
f
>>>HHG = llp.parse('(H -o (H -o G))')
>>>ff = HHG.applyto(f)
>>>print ff
(-o H (-o H G) (-o (-o g G) G (-o g f)) [(G, f)]) [(G, f), (H, f)]
>>>print ff.simplify().infixify()
(f -o f)
```

1.3 Glue Formulas

In chapter 11, we introduced the idea that words can be associated with meaning terms stated in λ-calculus. For example, the verb 'walk' can be represented as a function from a value of type **Ind** to a value of type **Bool**, λx.walk(x). Glue semantics extends this system by dictating that a Glue Formula is comprised of a meaning term and a "Glue Term". The Glue Term is a linear logic statement that constrains the ways that the Glue Formula may combine with other glue formulas. For the verb "walks", the glue statement would be '(g $\multimap$ f)' where 'g' represents the verb's subject (an Ind) and 'f' represents the verb's clause. Thus the Glue Formula for the verb 'walk' is a pair 'λx.(walk x) : (g $\multimap$ f)'.

To show how this Glue Formula is used, let us assume that a proper noun, such as 'John', is represented with the Glue Formula 'John : g' since John is of the type **Ind**. If we want to combine these formulas to get the meaning of the sentence "John walks" we will do so by allowing the glue terms to dictate how the meaning terms will be applied. The glue terms can be combined as a proof of 'f', which represents the entire sentence. This is shown as (7):

(7) $g, (g \multimap f) \vdash f$

In order to produce the reading for the sentence from this linear logic proof, we will have to attach the meaning terms to the glue terms and apply them appropriately at each step of the proof. The rule for applying one glue formula to another is shown in (8):

$$(8) \quad \frac{\phi : \mathrm{A} \qquad \psi : (\mathrm{A} \multimap \mathrm{B})}{\psi(\phi) : \mathrm{B}}$$

Therefore, all together we have the following proof that produces a semantic meaning for the sentence "John walks" in (9):

$$(9) \quad \frac{\mathrm{John} : \mathrm{g} \qquad \lambda x.(\mathrm{walks}\ x) : (\mathrm{g} \multimap \mathrm{f})}{\lambda x.(\mathrm{walks}\ x)(\mathrm{John}) : \mathrm{f}}$$

And 'λx.(walks x)(John)' β-reduces to '(walk John)'.

1.3.1 Glue Formulas in NLTK

The tools available for using Glue Semantics are found in the file "glue.py". The GlueFormula class is used for glue formulas. It contains an appyto() method for performing applications on other GlueFormulas.

```
>>>from nltk_lite.semantics import glue
>>>walks = glue.GlueFormula('\\x.(walks x)', '(g -o f)')
>>>walks
\x.(walks x) : (-o g f)
>>>john = glue.GlueFormula('John', 'g')
>>>john
John : g
>>>john_walks = walks.applyto(john)
>>>john_walks
(\x.(walks x) John) : (-o g f g)
>>>john_walks.simplify()
(walks John) : f
```

1.4 Syntax-Semantics Interface

An appropriate next question would ask how we determine which propositions to use in the glue terms. For example, in (9), why is 'John' paired with 'g' and 'walks' with '(g $\multimap$ f)'? The answer is that 'walks' is an intransitive verb and the meaning of an intransitive verb is a function that takes the verb's subject as input and returns the truth value of the verb's clause. In our example, 'g' is the proposition representing the verb's subject, so it is natural that it is the antecedent in the verb's glue term.

The traditional way that syntactic relationships are shown in Glue literature is with Lexical Functional Grammar [Dal01]. The sentence "John walks" would be represented as (10):

$$(10) \quad \mathrm{f}{:}\begin{bmatrix} \mathrm{PRED} & \text{`walks}\langle\mathrm{SUBJ}\rangle\text{'} \\ \mathrm{SUBJ} & \mathrm{g}{:}\begin{bmatrix} \mathrm{PRED} & \text{`John'} \end{bmatrix} \end{bmatrix}$$

This style of marking syntactic structure allows us to clearly see that 'f' represented the entire sentence and that 'g' represents the subject of the sentence. Let us compare this to the representation and subsequent proof (with β-reductions done as we progress) of the sentence "John sees Mary", shown as (11) and (12) respectively:

$$(11) \quad \mathrm{f}{:}\begin{bmatrix} \mathrm{PRED} & \text{`walks}\langle\mathrm{SUBJ, OBJ}\rangle\text{'} \\ \mathrm{SUBJ} & \mathrm{g}{:}\begin{bmatrix} \mathrm{PRED} & \text{`John'} \end{bmatrix} \\ \mathrm{OBJ} & \mathrm{h}{:}\begin{bmatrix} \mathrm{PRED} & \text{`Mary'} \end{bmatrix} \end{bmatrix}$$

(12) $$\dfrac{\text{Mary} : \text{h} \qquad \dfrac{\text{John} : \text{g} \qquad \lambda x.\lambda y.(\text{see } x\ y) : (\text{g} \multimap (\text{h} \multimap \text{f}))}{\lambda y.(\text{see John } y) : (\text{h} \multimap \text{f})}}{(\text{see John Mary}) : \text{f}}$$

In this case the verb is transitive, so it must have a subject ('John') and an object ('Mary').

1.4.1 Transitive Verbs in NLTK

```
>>>from nltk_lite.semantics import glue
>>>john = glue.GlueFormula('John', 'g')
>>>mary = glue.GlueFormula('Mary', 'h')
>>>sees = glue.GlueFormula('\\x y.(sees x y)', '(g -o (h -o f))')
>>>john_sees = sees.applyto(john)
>>>john_sees.simplify().infixify()
\y.(sees John y) : (h -o f)
>>>john_sees_mary = john_sees.applyto(mary)
>>>john_sees_mary.simplify().infixify()
(sees John Mary) : f
```

Also, notice what happens if we try to apply the individuals in the wrong sequence:

```
>>>sees.applyto(mary)
LinearLogicApplicationError: \x y.(sees x y) : (-o g (-o h f)) applied to Mary : h
```

1.5 Quantification

As was noted in chapter 11, a quantifier is of type '((**Ind** → **Bool**) → ((**Ind** → **Bool**) → **Bool**))'. A quantifier can combine with a noun, which is of type '**Ind** → **Bool**', to give a quantified noun, which is of type '((**Ind** → **Bool**) → **Bool**)'. A quantified noun can then be combined with a verb to make a clause.

For reference, let us give the LFG f-structure for the sentence "a man walks":

(13) $$\text{f:}\begin{bmatrix} \text{PRED} & \text{`walks}\langle\text{SUBJ}\rangle\text{'} \\ \text{SUBJ} & \text{g:}\begin{bmatrix} \text{PRED} & \text{`man'} \\ \text{SPEC} & \text{`a'} \end{bmatrix} \end{bmatrix}$$

We will examine the noun first. The meaning of a noun can be viewed as a function from an individual to a truth value. For example, the glue formula for the word "man" is given in (14):

(14) $\lambda x.(\text{man } x) : (\text{gv} \multimap \text{gr})$

The meaning term tells us that "man" is a function that takes an individual as input and returns a truth value for whether that individual is a man. The glue term naturally mirrors the single-argument-function pattern of the meaning term. Here we have *gv*, which is *g*'s VAR-value, and *gr*, which is *g*'s RESTR-value. We can see in (15) how a noun combines with a quantifier to create a quantified noun phrase.

(15) $$\dfrac{\lambda x.(\text{man } x) : (\text{gv} \multimap \text{gr}) \qquad \lambda P.\lambda Q.\text{some } x.((P\ x) \text{ and } (Q\ x)) : ((\text{gv} \multimap \text{gr}) \to ((\text{g} \multimap \text{G}) \multimap \text{G}))}{\lambda Q.\text{some } x.((man\ x) \text{ and } (Q\ x)) : ((\text{g} \multimap \text{G}) \multimap \text{G})}$$

At this point we must pause to discuss the glue proposition 'G'. In the glue for this quantified noun phrase, 'G' is a variable[1] standing for an expression of type **Bool**. As one would expect from a variable, it may bind with any proposition of type **Bool** in the course of a proof. We can now see how the result from (15) can be combined with the word "walks" to give the reading of the entire sentence. The proof is given in (16).

(16) $$\dfrac{\lambda x.(\text{walks } x) : (\text{g} \multimap \text{f}) \qquad \lambda Q.\text{some } x.((man\ x) \text{ and } (Q\ x)) : ((\text{g} \multimap \text{G}) \multimap \text{G})}{\text{some } x.((man\ x) \text{ and } (walks\ x)) : \text{f}}$$

[1]We will always use capital letters to distinguish linear logic variables.

An existential quantifier such as “a” or “some” is represented as (17). A universal quantifier such as “all” or “every” is represented as (18).

(17) $\lambda P.\lambda Q.\text{some } x.((P\ x) \text{ and } (Q\ x)) : ((\text{gv} \multimap \text{gr}) \multimap ((\text{g} \multimap \text{G}) \multimap \text{G}))$

(18) $\lambda P.\lambda Q.\text{all } x.((P\ x) \text{ implies } (Q\ x)) : ((\text{gv} \multimap \text{gr}) \multimap ((\text{g} \multimap \text{G}) \multimap \text{G}))$

1.5.1 Quantification in NLTK

```
>>>a = glue.GlueFormula('\\P Q.some x.((P x) and (Q x))', '((gv -o gr) -o ((g -o G) -o G))')
>>>man = glue.GlueFormula('\\x.(man x)', '(gv -o gr)')
>>>walks = glue.GlueFormula('\\x.(walks x)', '(g -o f)')
>>>a_man = a.applyto(man)
>>>print a_man.simplify().infixify()
\Q.some x.((man x) and (Q x)) : ((g -o G) -o G)
>>>a_man_walks = a_man.applyto(walks)
>>>print a_man_walks.simplify().infixify()
some x.((man x) and (walks x)) : f
```

1.6 Semantic Ambiguity

Semantic ambiguity is ambiguity that arises from semantics, even when the syntax is unambiguous. An example of a semantically ambiguous sentence is (1), repeated here as (19). It is ambiguous because it can be interpreted as either (20) or (21).

(19) Every girl chases a dog.

(20) all x.((girl x) implies some y. ((dog y) and (chase x y)))

(21) some y.((dog y) and all x. ((girl x) implies (chase x y)))

Glue semantics will allow us to generate both of these readings through the same proof strategies. But first, we will have to introduce a new proof rule. The rule in (22) allows us to create a unique[2] temporary hypothesis and then to abstract it away.

(22)
$$\dfrac{\begin{array}{c} x : [\text{A}]_i \\ \vdots \\ \psi : \text{B} \end{array}}{\lambda x.\psi : (\text{A} \multimap \text{B})}\ \text{ABS}_i$$

So, to generate readings for (19), we start with a list of premises:

(23) **[every]** $\lambda P.\lambda Q.\text{all } x.((P\ x) \text{ implies } (Q\ x)) : ((\text{gv} \multimap \text{gr}) \multimap ((\text{g} \multimap \text{G}) \multimap \text{G}))$

(24) **[girl]** $\lambda x.(\text{girl } x) : (\text{gv} \multimap \text{gr})$

(25) **[chases]** $\lambda x.\lambda y.(\text{chases } x\ y) : (\text{g} \multimap (\text{h} \multimap \text{f}))$

(26) **[a]** $\lambda P.\lambda Q.\text{some } x.((P\ x) \text{ and } (Q\ x)) : ((\text{hv} \multimap \text{hr}) \multimap ((\text{h} \multimap \text{H}) \multimap \text{H}))$

(27) **[dog]** $\lambda x.(\text{dog } x) : (\text{hv} \multimap \text{hr})$

It is easy to see that “every” can only combine with “girl” and “a” only with “dog”. We perform these combinations to generate both quantified nouns.

(28) **[every-girl]** $\lambda Q.\text{all } x.((\text{girl } x) \text{ implies } (Q\ x)) : ((\text{g} \multimap \text{G}) \multimap \text{G})$

[2] The index i in the proof must always be fresh to ensure uniqueness

(29) **[a-dog]** $\lambda Q.\text{some } x.((\text{dog } x) \text{ and } (Q\ x)) : ((\text{h} \multimap \text{H}) \multimap \text{H})$

Now we can generate exactly two readings from these two quantified nouns and the verb "chases". The proofs are detailed as (30) and (31):

(30)

$$\dfrac{\dfrac{\dfrac{\dfrac{x^1 : [\text{g}]_1 \qquad \lambda x.\lambda y.(\text{chases } x\ y) : (\text{g} \multimap (\text{h} \multimap \text{f}))}{\lambda y.(\text{chases } x^1\ y) : (\text{h} \multimap \text{f})} \qquad \lambda Q.\text{some } x.((\text{dog } x) \text{ and } (Q\ x)) : ((\text{h} \multimap \text{H}) \multimap \text{H})}{\text{some } x.((\text{dog } x) \text{ and } (\text{chases } x^1\ x)) : \text{f}}}{\lambda x^1.\text{some } x.((\text{dog } x) \text{ and } (\text{chases } x^1\ x)) : (\text{g} \multimap \text{f})}\,\text{ABS}_1 \qquad \lambda Q.\text{all } x.((\text{girl } x) \text{ implies } (Q\ x)) : ((\text{g} \multimap \text{G}) \multimap \text{G})}{\text{all } x.((\text{girl } x) \text{ implies } (\text{some } z^1.((\text{dog } z^1) \text{ and } (\text{chases } x\ z^1)))) : \text{f}}$$

(31)

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{x^2 : [\text{h}]_2 \qquad \dfrac{x^1 : [\text{g}]_1 \qquad \lambda x.\lambda y.(\text{chases } x\ y) : (\text{g} \multimap (\text{h} \multimap \text{f}))}{\lambda y.(\text{chases } x^1\ y) : (\text{h} \multimap \text{f})}}{(\text{chases } x^1\ x^2) : \text{f}}}{\lambda x^1.(\text{chases } x^1\ x^2) : (\text{g} \multimap \text{f})}\,\text{ABS}_1 \qquad \lambda Q.\text{all } x.((\text{girl } x) \text{ implies } (Q\ x)) : ((\text{g} \multimap \text{G}) \multimap \text{G})}{\text{all } x.((\text{girl } x) \text{ implies } (\text{chases } x\ x^2)) : \text{f}}}{\lambda x^2.\text{all } x.((\text{girl } x) \text{ implies } (\text{chases } x\ x^2)) : (\text{h} \multimap \text{f})}\,\text{ABS}_2 \qquad \lambda Q.\text{some } x.((\text{dog } x) \text{ and } (Q\ x)) : ((}{\text{some } x.((\text{dog } x) \text{ and } (\text{all } z^1.((\text{girl } z^1) \text{ implies } (\text{chases } z^1\ x)))) : \text{f}}$$

1.6.1 Semantic Ambiguity in NLTK

To generate the readings of the sentence "Every girl chases a dog", we must generate the quantified noun phrases 'every girl' and 'a dog' first, along with the verb 'chases'.

```
>>>every = glue.GlueFormula('\\P Q.all x.((P x) implies (Q x))', '((gv -o gr) -o ((g -o G) -o G))')
>>>girl = glue.GlueFormula('\\x.(girl x)', '(gv -o gr)')
>>>every_girl = every.applyto(girl)
>>>print every_girl.simplify().infixify()
\Q.all x.((girl x) implies (Q x)) : ((g -o G) -o G)
>>>chases = glue.GlueFormula('\\x y.(chases x y)', '(g -o (h -o f))')
>>>print chases.infixify()
\x y.(chases x y) : (g -o (h -o f))
>>>a = glue.GlueFormula('\\P Q.some x.((P x) and (Q x))', '((hv -o hr) -o ((h -o H) -o H))')
>>>dog = glue.GlueFormula('\\x.(dog x)', '(hv -o hr)')
>>>a_dog = a.applyto(dog)
>>>print a_dog.simplify().infixify()
\Q.some x.((dog x) and (Q x)) : ((h -o H) -o H)
```

Because the rest of the assembly requires the abstraction rule (22), we will need a new technique to achieve the functionality of this rule. We will do this by creating a hypothesis glue formula whose meaning term is a variable and whose glue term is the glue expression needed. We can then abstract this hypothesis away using the method lambda_abstract() which takes the hypothesis to be abstracted as its argument.

```
>>>x1 = glue.GlueFormula('x1', 'A')
>>>psi = glue.GlueFormula('\\x.(psi x)', '(A -o B)')
>>>psi_x1 = psi.applyto(x1)
>>>print psi_x1.simplify()
(psi x1) : B
>>>psi2 = psi_x1.lambda_abstract(x1)
>>>print psi2
\x1.(\x.(psi x) x1) : (-o A (-o A B A))
>>>print psi2.simplify()
\x1.(psi x1) : (-o A B)
```

We can now use this technique to generate the readings for "Every girl chases a dog" from the quantified nouns and the verb. The first will be modeled after (30):

```
>>>x1 = glue.GlueFormula('x1', 'g')
>>>x1_chases = chases.applyto(x1)
>>>print x1_chases.simplify().infixify()
\y.(chases x1 y) : (h -o f)
>>>x1_chases_a_dog = a_dog.applyto(x1_chases)
>>>print x1_chases_a_dog.simplify().infixify()
some x.((dog x) and (chases x1 x)) : f
>>>chases_a_dog = x1_chases_a_dog.lambda_abstract(x1)
>>>print chases_a_dog.simplify().infixify()
\x1.some x.((dog x) and (chases x1 x)) : (g -o f)
>>>every_girl_chases_a_dog = every_girl.applyto(chases_a_dog)
>>>print every_girl_chases_a_dog.simplify().infixify()
all x.((girl x) implies some z1.((dog z1) and (chases x z1))) : f
```

The second will be modeled after (31):

```
>>>x1 = glue.GlueFormula('x1', 'g')
>>>x1_chases = chases.applyto(x1)
>>>print x1_chases.simplify().infixify()
\y.(chases x1 y) : (h -o f)
>>>x2 = glue.GlueFormula('x2', 'h')
>>>x1_chases_x2 = x1_chases.applyto(x2)
>>>print x1_chases_x2.simplify().infixify()
(chases x1 x2) : f
>>>chases_x2 = x1_chases_x2.lambda_abstract(x1)
>>>print chases_x2.simplify().infixify()
\x1.(chases x1 x2) : (g -o f)
>>>every_girl_chases_x2 = every_girl.applyto(chases_x2)
>>>print every_girl_chases_x2.simplify().infixify()
all x.((girl x) implies (chases x x2)) : f
>>>every_girl_chases = every_girl_chases_x2.lambda_abstract(x2)
>>>print every_girl_chases.simplify().infixify()
\x2.all x.((girl x) implies (chases x x2)) : (h -o f)
>>>every_girl_chases_a_dog = a_dog.applyto(every_girl_chases)
>>>print every_girl_chases_a_dog.simplify().infixify()
some x.((dog x) and all z1.((girl z1) implies (chases z1 x))) : f
```

1.7 Computational Issues in Glue Semantics

1.7.1 Generation of Glue Formulas

To generate the glue formulas that will be used in the proof, we must first parse the sentence for syntax (see Chapters 7 and 8). The glue module includes a class FStructure that is created from a parse tree.

```
>>>from nltk_lite.parse import GrammarFile
>>>from nltk_lite import tokenize
>>>grammar = GrammarFile.read_file(’grammar.cfg’)
>>>parsetrees = grammar.earley_parser(0).get_parse_list(list(tokenize.whitespace(’John sees a woman’)))
>>>from nltk_lite.semantics import glue
>>>glue_dict = glue.GlueDict()
>>>glue_dict.read_file(’glue.cfg’)
>>>fstruct = glue.FStructure(parsetrees[0][0], [0])
>>>print fstruct
f:[pred ’sees’
  subj g:[pred ’John’]
  obj h:[pred ’woman’
       spec ’a’]]
>>>for gf in fstruct.to_glueformula_list(glue_dict): print gf
\x y.(sees x y) : (-o g (-o h f))
John : g
\x.(woman x) : (-o hv hr)
\P Q.some x.(and (P x) (Q x)) : (-o (-o hv hr) (-o (-o h H0) H0))
```

1.7.2 Horn Clauses

The problem of having the computer automatically assemble a sentence's glue premises into a proof becomes much more difficult with the rule given in (22) because the rule opens up the possibility of an infinite number of proofs for any valid set of premises. This occurs because we are able to introduce and retract hypotheses at will, whether they are required or not. So from the premises ’g’ and ’(g ⊸ f)’ we can generate the following proofs (32)-(34):

(32) $$\frac{g \qquad (g \multimap f)}{f}$$

(33) $$\frac{g \qquad \dfrac{\dfrac{[g]_1 \qquad (g \multimap f)}{f}}{(g \multimap f)}\,\text{ABS}_1}{f}$$

(34) $$\frac{g \qquad \dfrac{\dfrac{[g]_2 \qquad \dfrac{\dfrac{[g]_1 \qquad (g \multimap f)}{f}}{(g \multimap f)}\,\text{ABS}_1}{f}}{(g \multimap f)}\,\text{ABS}_2}{f}$$

It should be clear that we could generate an infinite number of proofs following this pattern of hypothesizing and retracting variables. It should also be clear that a single proof could involve an infinite number of these actions, and thus cause “proving” to take forever. This is a major problem because in glue semantics we want to generate all possible proofs so that we can have all possible readings since each new proof could potentially give a different reading.

The solution to this problem comes from [GL98]. The basic idea is to “compile” linear logic formulas into “skeletons”, which are of the form $(g_1 \multimap (g_2 \multimap (\ldots \multimap f)))$ where g_i and f are propositions. By doing this we are generating new premises for all of the hypotheses that we would have needed to generate during the proof. However, by compiling, we ensure that we will never need to make any hypotheses, and can therefore rid ourselves of the problematic rule (22).

The algorithm for linear logic compilation from [Lev07] is given as Figure 1.

The compilation method changes our representation of a glue formula slightly because it affixes a **list of indices** to every glue formula. When a premise is created, this list is instantiated with one unique index. Additionally, every atomic linear logic expression will have its own **list of dependencies**.

```
function compile-glue-formula($\psi : X$)
    $\langle Y, \Gamma\rangle$ := compile-pos($X$)
    return $\{\psi : Y : \{i\}\} \cup \Gamma$ for a fresh index $i$

function compile-pos(X) where $X$ atomic
    return $\langle X_{[]}, \emptyset\rangle$

function compile-pos($X \multimap Y$)
    $\langle \delta, \Delta\rangle$ := compile-pos($Y$)
    $\langle \gamma, \Gamma\rangle$ := compile-neg($X$) ($\gamma$ will be atomic)
    return $\langle (\gamma \multimap \delta), \Gamma \cup \Delta\rangle$

function compile-neg(X) where $X$ atomic
    return $\langle X_{[]}, \emptyset\rangle$

function compile-neg($X \multimap Y$)
    $\langle \delta_L, \Delta\rangle$ := compile-neg($Y$) ($\delta$ will be atomic)
    $\langle \gamma, \Gamma\rangle$ := compile-pos($X$)
    return $\langle \delta_{[i|L]}, \{v_i : \gamma : \{i\}\} \cup \Gamma \cup \Delta\rangle$ for a fresh index $i^\dagger$
```
$^\dagger[i|L]$ means a list whose head is i and whose tail is L

Figure 1: Linear Logic Compilation Algorithm

To turn our premises into "skeletons", we must get rid of any non-atomic antecedents[3]. We will use (35) as an example.

(35) 'm : ((A $\multimap$ B) $\multimap$ C)'

To compile (35) we would turn 'A' into its own premise. Since another premise 'A' might exist already, we must ensure that **this** 'A' is the one treated as the antecedent to 'B'. We do this by adding the newly created 'A' premise's index to 'B's list of dependencies. Compilation of (35) will return glue formulas (37) and (36).

(36) '$v1$: A : {1}'

(37) 'm : (B$_{[1]}$ $\multimap$ C) : {2}'

In the NLTK lite package, glue formula compilation can be performed using the 'compile()' method:

```
>>>from nltk_lite.semantics import glue
>>>g = glue.GlueFormula('m', '((A -o B) -o C)')
>>>gc = g.compile()
>>>print gc[0]
m : (-o B[1] C) : {2}
>>>print gc[1]
v1 : A : {1}
```

1.7.3 Horn Clause Application

Naturally, we will have to incorporate the indices and dependencies into our application rule from (8). The new rule is shown below as (38):

$$(38)\qquad \frac{\phi : \mathrm{A} : \Gamma \qquad \psi : (\mathrm{A}_L \multimap \mathrm{B}) : \Delta}{(\psi\ \lambda V i_1, \ldots, \lambda V i_n.\phi) : \mathrm{B} : \Gamma \cup \Delta} \text{ provided } L \subset \Gamma \text{ and } L = [i_1, \ldots, i_n]$$

[3]We must also move through the formula recursively to ensure that all of its subformulas are also skeletons.

You will notice that the set of indices of the conclusion is the sets of indices of both premises combined. This means that a glue formula's set of indices is the set of premises that were consumed to generate that formula. It should be noted that Γ and Δ will always be disjoint since the rules of linear logic say that no premise may be used more than once. We also impose the condition that 'α' must be a subset of 'Γ'. This will ensure that the correct premises were used to generate the 'A' premise used in the application.

The requirement that L be a subset of Γ means that for the application to be successful, ϕ must have already incorporated all of the dependencies required by L.

The meaning term of the conclusion formula of (38), 'ψ' is not applied directly to 'ϕ'. It is applied to a term that is generated by adding λ-abstractions of each meaning term of an glue formula indexed by an index in 'L'. It is important to maintain the order of the elements of 'L' since the order of the abstractions does matter. When $n = 0$, $\lambda vi_1, \ldots, \lambda vi_n.\phi$ is simply ϕ.

The example below demonstrates how the NLTK handles horn clause applications:

```
>>>from nltk_lite.semantics import glue
>>>a_man = glue.GlueFormula('\\Q.some x.((man x) and (Q x))', '((g -o G) -o G)')
>>>walks = glue.GlueFormula('\\x.(walks x)', '(g -o f)')
>>>amc = a_man.compile([1])
>>>g2 = amc[0]
>>>g1 = amc[1]
>>>g3 = walks.compile([3])[0]
>>>g1
v1 : g : {1}
>>>g2
\Q.some x.(and (man x) (Q x)) : (-o G[1] G) : {2}
>>>g3
\x.(walks x) : (-o g f) : {3}
>>>g13 = g3.applyto(g1)
>>>g13
(\x.(walks x) v1) : (-o g f g) : {1, 3}
>>>g13.simplify()
(walks v1) : f : {1, 3}
>>>g123 = g2.applyto(g13)
>>>g123.infixify()
(\Q.some x.((man x) and (Q x)) \v1.(\x.(walks x) v1)) : (G[1] -o G (g -o f g)) [(G, f)] : {1, 2, 3}
>>>g123.simplify().infixify()
some x.((man x) and (walks x)) : f : {1, 2, 3}
```

1.7.4 Machine Derivation

The algorithm we will use to have the computer automatically assemble glue proofs is from [Lev07]. The algorithm starts with a list of compiled glue formulas, called the *agenda*. When it begins, it initializes two dictionaries, one to hold atomic formulas, and the other to hold non-atomic formulas (implications). The algorithm then iterates as follows:

```
While the agenda is not empty, remove an element cur.
   If cur is non-atomic:
      For each formula atomic in the atomics dictionary that cur can be applied to:
         Apply cur to atomic and place the result in the agenda
      Place cur in the nonatomics dictionary
   Else cur is atomic:
      For each formula nonatomic in the nonatomics dictionary that can be applied to cur:
         Apply nonatomic to cur and place the result in the agenda
      Place cur in the atomics dictionary
Return the list of elements in the dictionaries with a complete set of indices
```

The algorithm is given as python code in Figure 2. The argument passed to the function get_readings is a list of compiled glue formulas.

1.8 Further Reading

See http://www-csli.stanford.edu/~iddolev/glue_bibliography.html for a comprehensive bibliography of glue semantics literature put together by Iddo Lev.

References

[Dal01] Mary Dalrymple. *Lexical Functional Grammar*, volume 34 of *Syntax and Semantics.* Academic Press, New York, 2001.

[GL98] Vineet Gupta and John Lamping. Efficient linear logic meaning assembly. In *Proc. of COLING/ACL98*, 1998.

[Lev07] Iddo Lev. *Packed Computation of Exact Meaning Representations.* PhD thesis, Stanford University, 2007.

```
def get_readings(agenda):
    from nltk_lite.semantics import glue
    readings = []
    agenda_len = length(agenda)
    atomics = dict()
    nonatomics = dict()
    while agenda: # is not empty
        cur = agenda.pop()
        # if agenda.glue is non-atomic
        if isinstance(cur.glue.simplify(), linearlogic.ApplicationExpression):
            for key in atomics:
                if cur.glue.simplify().first.second.can_unify_with(key, cur.glue.varbindings):
                    for atomic in atomics[key]:
                        if cur.indices.intersection(atomic.indices):
                            continue
                        else: # if the sets of indices are disjoint
                            try:
                                agenda.append(cur.applyto(atomic))
                            except linearlogic.LinearLogicApplicationError:
                                pass
            try:
                nonatomics[cur.glue.simplify().first.second].append(cur)
            except KeyError:
                nonatomics[cur.glue.simplify().first.second] = [cur]

        else: # else agenda.glue is atomic
            for key in nonatomics:
                for nonatomic in nonatomics[key]:
                    if cur.glue.simplify().can_unify_with(key, nonatomic.glue.varbindings):
                        if cur.indices.intersection(nonatomic.indices):
                            continue
                        else: # if the sets of indices are disjoint
                            try:
                                agenda.append(nonatomic.applyto(cur))
                            except linearlogic.LinearLogicApplicationError:
                                pass
            try:
                atomics[cur.glue.simplify()].append(cur)
            except KeyError:
                atomics[cur.glue.simplify()] = [cur]

    for entry in atomics:
        for gf in atomics[entry]:
            if len(gf.indices) == agenda_length:
                readings.append(gf.meaning)
    for entry in nonatomics:
        for gf in nonatomics[entry]:
            if len(gf.indices) == agenda_length:
                readings.append(gf.meaning)
    return readings
```

Figure 2: Machine Deduction Code