

Yet Another Attempt to Explain Glue

Avery D Andrews

ANU, May 2009

‘Glue semantics’ is certainly the nearest thing there is to an ‘official’ theory of compositional semantics for LFG, but suffers from the problem that a great many practitioners of LFG syntax profess not to understand it. This is unfortunate for a variety of reasons, not the least of which is that with the addition of glue, LFG finally attains Richard Montague’s standard of adequacy for grammatical theories, that the overt forms of sentences be explicitly connected to structures that can support a mathematical definition of entailment.¹ Conceptually, this puts LFG on a par with current fully formalized theories such as HPSG and Type-Logical Grammar, but the effect is certainly diluted if most LFG practitioners don’t understand how it works.

Glue furthermore provides an additional level of representation, which can be strictly binary branching if this is desired, which can be used to account for various phenomena that don’t fit easily into the f-structure format. Scope of quantifiers, negation, various adverbs, etc. are some classic examples, more recent ones are Andrews’ 2007b proposals about complex predicates, where glue is used to implement Alsina’s (1997) concept of ‘predicate composition’, and Sadler and Nordlinger’s (2008) work on the structure of NPs in Australian languages, where glue is used to differentiate interpretational differences that don’t appear to be supported in the f-structures. This might provide a bridge between LFG and other frameworks such as the Minimalist Program, where binary-branching trees that reflect semantic composition play a central role.

This presentation of glue will at first be streamlined and modified in certain respects from the standard one, as presented for example in Dalrymple (2001) and Asudeh (2004), but the differences will be explained later. Sections (1-8) are hopefully reasonably self-contained, and even attempt to introduce some basic ideas of formal semantics to people who aren’t familiar with them, but 10 in particular is at a significantly higher level, and after looking at 9 it is probably time to take on some other glue literature before tackling 10. Asudeh (2005a) would be a good choice, alongside the longer Dalrymple and Asudeh works cited above. Several aspects of the presentation used here, ‘prefab glue’, are discussed more formally in Andrews (2008a).

1 Semantic (Combinatorial) Types

We start by considering the nature of the ‘semantic types’ used in formal semantics, which are fundamental to glue. These can be thought of as a system of ontological categories that limit the ways in which concepts can coherently be combined.² Since the main function of these types appears to be to control combinations, rather than

¹Which is, it seems to me, the empirically straightforward aspect of the program of ‘truth conditional semantics’, whatever one thinks about the philosophical issues.

²See Casadio (1988) for a discussion of the historical background of type theory.

actually say anything about what the meanings are, it would also be sensible to call them ‘combinatory types’.³

Theories of semantic/combinatory types start with the idea of an inventory of ‘basic types’, together with some operations for combining these to produce complex/derived types. In linguistics, it is common to recognize at least two basic types, entities, which we’ll symbolize as e , and ‘propositions’, which we’ll follow Pollard (2007) by symbolizing as p (t is the more common choice). There might be many more types, for reasons discussed in many works by Jackendoff, e.g. Jackendoff (2002), such as ev for events, loc for locations, pth for paths, and deg for degrees of magnitude), or only one (Partee 2006), which is effectively none. But for slightly technical reasons, not covered here,⁴ LFG glue is generally thought to need at least the distinction between e and p , so we’ll follow the tradition and assume these two.

From the basic types, we can construct an infinite variety of additional types by means of ‘type-constructors’, the most important of which is the ‘implicational’, which combines two types a and b to produce the type $a \rightarrow b$.⁵ This is the type of something which, if you combine it with something of type a , then you get something of type b (hence the name, ‘implicational’). A very simple example is the type of intransitive verbs, such as *yell* or *stumble*. These, if combined with something designating an entity (*Bill*, or *the pony*) produce a proposition, so they are of type $e \rightarrow p$. Another simple one is negation: since a negative turns a proposition into another proposition (true where the first proposition is false, false where the first proposition is true), its type will be $p \rightarrow p$. The implicational types can in fact be regarded as mathematical functions, which operate on a meaning of one type, to produce a different meaning of some other type. For this reason, they can also be called ‘functional types’.

What about transitive verbs, such as *likes*? Here we use an idea developed independently by the logicians M. Schönfinkel and H.B. Curry (Schönfinkel first), and introduced into linguistics by Richard Montague (following Lambek (1965) and various other sources), that in a sentence such as *Bert likes Ernie*, one can regard the traditional ‘predicate’ or contemporary ‘VP’ *likes Ernie* as being of type $e \rightarrow p$. So what about *likes* itself? Well, this is something to which one supplies an e , and gets an $e \rightarrow p$, so it can be put into the type $e \rightarrow (e \rightarrow p)$.

This is really a rather clever idea. The Ancient Greeks, such as Aristotle, had a pretty good grasp of the workings of the types e and $e \rightarrow p$ (and some others), but never came up with $e \rightarrow (e \rightarrow p)$, so their logic stalled (for more than 2000 years) on the question of what to do about transitive verbs. So they had a good story about deductions from intransitive clauses (the sentence below the line is supposed to follow from the ones above it):

³Not least because they have quite a lot to do with a subject called ‘combinatory logic’.

⁴But discussed in Andrews (2008a).

⁵In systems where there is only one type-constructor, this is often omitted, and $\langle a, b \rangle$ is written rather than $a \rightarrow b$.

- (1) All men are mortal
 Socrates is a man

 Socrates is mortal

But their system couldn't account for a deduction involving both the subject and object of a transitive verb, such as:

- (2) Every American likes every Muppet
 George is an American
 Ernie is a Muppet

 George likes Ernie

This problem wasn't overcome until the late 19th century (by C.S. Peirce and Gottlob Frege, working independently), using a significantly different treatment of transitive verbs, which people who haven't done 'Predicate Calculus' in a basic course in logic shouldn't worry about now.⁶

We can use this idea to get various more types; for example, a ditransitive verb such as *give* would be $e \rightarrow (e \rightarrow (e \rightarrow p))$, while a 'propositionally ditransitive' verb such as *tell* in *John told Mary that she was sick* would be of type $p \rightarrow (e \rightarrow (e \rightarrow p))$.

The parenthesis-nests associated with multi-argument verbs can get confusing, so people often follow a convention of omitting rightmost pairs of parentheses, so that the three types above become $e \rightarrow e \rightarrow p$, $e \rightarrow e \rightarrow e \rightarrow p$, and $p \rightarrow e \rightarrow e \rightarrow p$. For computer programmers, these conventions have the pleasant property of being identical to the Haskell notation for declaring the types of functions.

A significant issue is that the exact order of the arguments is 'logically arbitrary'. That is, there's no reason in terms of pure formal semantics why we should supply the p argument of *tell* first, and then the others. However a long tradition of work on the 'thematic role hierarchy' in linguistics (Jackendoff (1973) is an especially important early example) provides evidence for a non-arbitrary arrangement of the arguments, so that the Theme (thing conveyed) would be innermost, then Recipient, and finally Agent. Marantz (1984) argues that this hierarchy should be thought as a matter of 'order of application', with the less active arguments applied first. I suggest that it would perhaps be better to see it as a hierarchy of tightness of structural bonding, 'earlier application' being tighter bonding. Regardless of how it's construed, it provides a non-arbitrary answer to how the arguments should be ordered, in most (but not all) cases.

⁶For those who have, however, the representations used here, called 'Curried', are equivalent to the ' n -place predicates' in first order logic, though it might take a beginner a certain amount of pondering to see this.

2 Expressions

The types are supposed to represent the possibilities for expressions to combine, so the next step is to provide some formulation of how to combine them. Implicational types are thought of as things with a ‘hole’ in them, of some type, into which something else of the same type is to fit, so what we want is an expression combining things of these two types into one, producing an expression of the appropriate resulting type. For example, combining something of type e with something of type $e \rightarrow p$ to produce something of type p . In logic, people are concerned with writing expressions as linear strings (‘formulas’), but for linguistics, we really only want the parse-trees for the formulas. The obvious tree for combining an e with an $e \rightarrow p$ is:

$$(3) \quad \begin{array}{c} p \\ \swarrow \quad \searrow \\ e \quad e \rightarrow p \end{array}$$

Well, obvious except for our decision to put the functional type second rather than first in the tree; as things work out, this seems to be on the whole best for layout, although the decision is arbitrary (and the trees could in fact be regarded as unordered).

And for a transitive verb, we’d have something like this:

$$(4) \quad \begin{array}{c} p \\ \swarrow \quad \searrow \\ e \quad e \rightarrow p \\ \quad \swarrow \quad \searrow \\ \quad e \quad e \rightarrow e \rightarrow p \end{array}$$

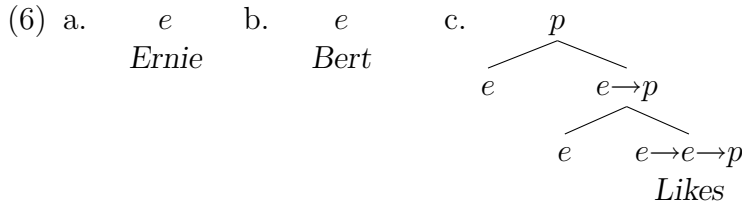
Following Marantz’s idea, the inner e position would be associated with the less active semantic role (for example, Liked), the outer one, the more active (Liker). This is in conformity with standard ideas about the organization of ‘argument-structure’, for example Alsina (1996), as discussed in Andrews (2007b).

To produce a standard ‘logical form’, we could write in the expressions under the leaf-nodes indicating their type, e.g.:

$$(5) \quad \begin{array}{c} p \\ \swarrow \quad \searrow \\ e \quad e \rightarrow p \\ Bert \quad \swarrow \quad \searrow \\ \quad e \quad e \rightarrow e \rightarrow p \\ \quad Ernie \quad Likes \end{array}$$

But our goal is not merely to represent combinations of meanings,⁷ but to also say something about how the one that is meant by the speaker can be figured out by the hearer, on the basis of the grammatical structure of the sentence.

So suppose we have a collection of words/meanings of various types. From the types alone, we can produce pieces of the structures they will fit into:



Rules for producing the tree-pieces from the types can be described as follows:

- (7) a. Starter: the starter node has the type as its ‘semantic type’, and the meaning as its ‘meaning-label’.
- b. Atom: if the current (initially, starter) node has atomic semantic type, the construction is finished.
- c. Implication: if the current node is of semantic type $a \rightarrow b$, make it the right-daughter of new node of semantic type b , with new left-daughter of semantic type a . Repeat (b-c) with the new mother as current (neither new node has a meaning label, although there is a way to assign meaning-labels to all nodes of fully assembled structures).

There is an important situation which these rules don’t cover, which we will get to later.

What we’ll do now is show how to assemble these ‘prefab’ pieces, so-called because they’re put into the structure rather than the type format before rather than after assembly, into full structures.

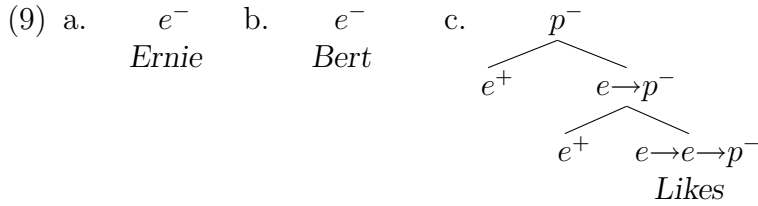
Happily, we don’t have to make it up by ourselves. There is a branch of logic concerned with ‘proof-nets for linear logic’, which addresses this issue, exactly. Even better, you don’t have to know anything about linear logic and proof-nets to learn the rules and use them (though you will, thereby, wind up having learned something about linear logic and proof-nets).

One basic ingredient in the recipe is a concept of ‘polarity’, which is used to control how the trees are assembled. The polarities are ‘negative’ and ‘positive’, and assigned to the tree-nodes by the following rules:

⁷Note the phrasing: we are not claiming to represent meanings, but only how they are to be combined, since the meanings themselves are not given, only their English glosses in a special font. This is standardly called ‘compositional semantics’, sometimes ‘combinatorial semantics’, as opposed to ‘lexical semantics’, or, one could say ‘full semantics’, in which both the lexical and syntactic contributions to meaning are treated.

- (8) a. Rightmost (meaning-bearing) leaf-nodes have negative polarity.
 b. If an implication has negative polarity, so does its mother, but its (left) sister has positive polarity.

These polarity assignments may seem a bit odd for the purposes of linguistics, and have occasionally been reversed in the LFG glue literature, but they are standard in logic, and it's probably best to follow that tradition here.



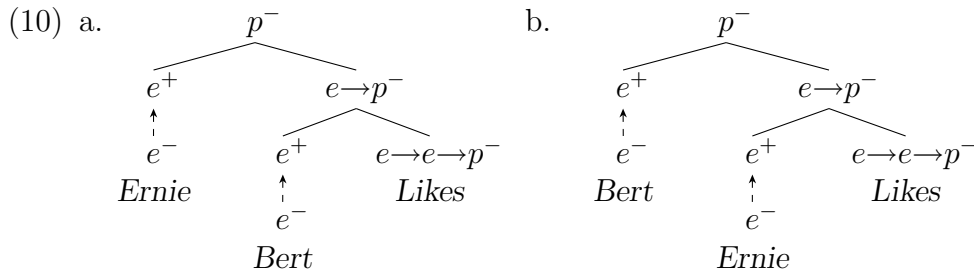
Note that (a) and (b) get negative polarity because they are rightmost (that they are also leftmost doesn't matter). It would be a good exercise to write out the tree for *tell* without the polarities, and then use the rule (8) to fill them in (answer in (99)).

So the next task is to describe a technique for assembling these and similar pieces.

What we're about to look at might seem unnecessarily complex, but its properties are motivated by the fact that it will be able to deal with somewhat more difficult situations we will encounter later, where naive, intuitive schemes are likely to be open to various kinds of abuse.

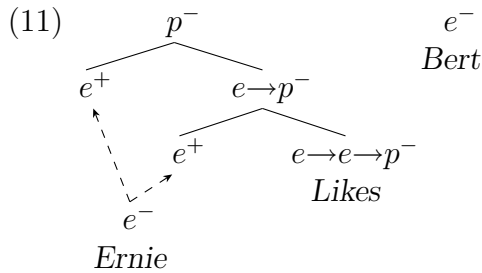
The basic idea is that we will assemble the pieces by running things called 'axiom links' from nodes with negative polarity to those with positive polarity that are labelled with a basic type. Nodes with basic type labels are called 'literals', and constitute most of the 'vertical frontier' of the tree, that is the nodes that either have no mother or no daughter (the negative literal is at the top, with no mother, the positive literals are the various non-rightmost leaves, with no daughters). The only vertical frontier node that doesn't have to be a literal is the one that carries the meaning, the rightmost daughter. The axiom links themselves will be depicted as dashed arrows.

What we intend is that there be two possible assemblies for the pieces in (9):



These can be converted into the earlier expression format of (5) by erasing the polarities, and ‘contracting’ the axiom links, that is, fusing the two nodes connected by an axiom-link into one.

Typical configurations that we would want to exclude are non-connected positive literals (as for example in (9), where no axiom-links have (yet) been added, and ‘multiple’ connections, such as:



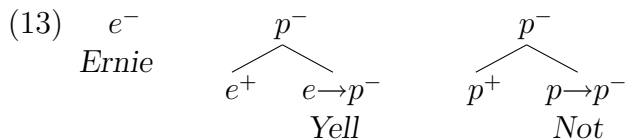
Here we want to reject the structure whether or not *Bert* is included or omitted. The left-hand tree might seem to be a good idea for the analysis of reflexive pronouns, but in fact, it doesn’t seem to work out to try to do this in the first stages of meaning-assembly.

The basic principles that rule out these and various other kinds of bad results are:

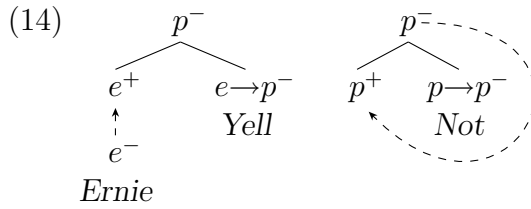
- (12) a. Matching: axiom-links go from negative to positive literals of the same type
 b. Monogamy: forming non-overlapping pairs
 c. One Left Over: with one negative polarity literal left over, and all other literals in axiom-linked pairs

These rules forbid all possibilities but those in (10) for assembling the pieces in (9). Observe that the idea of contracting along the axiom links wouldn’t produce very sensible results if we didn’t obey (a-b). However, deviously minded individuals can come up with some additional possibilities for certain other collections of pieces.

Consider for example some (hypothetical, oversimplified) component meanings for *Ernie doesn’t yell*:



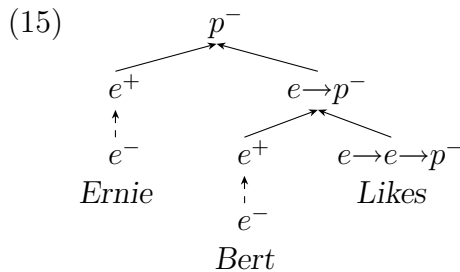
Here, in addition to the sensible assembly which you can hopefully come up with yourself (answer in (100)), there is also a perverse one that satisfies the conditions of (12):



Here we've plugged the negative 'result' p of the *Not*-piece into its positive 'argument' p . To rule this out definitively, we need an explicit principle.

The one that we will use is called the 'Correctness Criterion for proof-nets', which exists in a very large number of different-looking but equivalent formulations.⁸ The version we will use is based on de Groote (1999), and for the kinds of cases we've considered so far, can be impressionistically stated as: 'the assembled structure must be a tree'. But this is too vague for serious work, and also won't work in the most general case we would like to be able to deal with, so we need to define a few things more explicitly.

Technically, the assemblies we're making are 'graphs',⁹ which, for our purposes, can be thought as sets of 'nodes' (which can have various kinds of labels and other attributes) connected by 'directed arcs', which go from one node, the source of the arc, to another, its target. With the direction ('orientation') of the arcs indicated, assembly (10a) becomes:



The tree-nodes, with arcs directed as indicated, constitute a structure called the 'dynamic graph' in de Groote (1999). To formulate the Correctness Criterion, we need two more definitions:

- (16) a. The 'Principal Inputs' are the rightmost leaves (the nodes with meaning-names such as *Ernie* and *Like*; it doesn't matter if they're also leftmost).
- b. The 'Final Output' is the unlinked negative literal (there must be one and only one, according to the One Left Over principle of (12)).

⁸To which the most accessible general introduction is probably Moot (2002:ch.4), although this would still be pretty hard for people who don't know a reasonable amount about logic.

⁹In one of several meanings of the term that are used in mathematics. This one is unfortunately quite different from what you probably studied in secondary school algebra, but is one of the standard ones studied in 'Graph Theory'.

Now we can state the criterion as follows:

- (17) Correctness Criterion (clause 1): From every Principal Input, the dynamic graph must provide a path to the Final Output

This is ‘clause 1’ because we’ll be needing a second provision before too long.

You should try to work through that the sensible assemblies all satisfy (17), while (14) does not. Discussion of (15) and (14) in (101). An essential characteristic of this system is that it requires that each meaning introduced by the grammatical and lexical structure of a sentence be used once and once only in its interpretation. For example, if a sentence contains a negation, you can neither understand as positive by ignoring the negative, nor by interpreting twice as a double-negation¹⁰ (but overt multiple negations can be packed into one by LFG feature-unification, and thereby interpreted as a single positive).

We now have half of what will be a system for assembling ‘typed meanings’, that is, a combination of a meaning and its semantic type, into sensible combinations. But before producing the other half, we will look into how the grammatical structure of a sentence is to constrain its assembly.

3 Connecting to Grammar

Typed meanings already ‘know’ a fair amount about how they are to be put together. For example, there is only one way to assemble the pieces of (13). But for (9), there are two, so we must appeal to the syntax to account for why *Bert likes Ernie* and *Ernie likes Bert* aren’t both ambiguous. In LFG, these two sentences would differ in their f-structure:

$$(18) \text{ a. } \left[\begin{array}{l} \text{SUBJ } g: [\text{PRED 'Bert'}] \\ f: \text{PRED 'Like(SUBJ, OBJ)'} \\ \text{OBJ } h: [\text{PRED 'Ernie'}] \end{array} \right] \quad \text{b. } \left[\begin{array}{l} \text{SUBJ } g: [\text{PRED 'Ernie'}] \\ f: \text{PRED 'Like(SUBJ, OBJ)'} \\ \text{OBJ } h: [\text{PRED 'Bert'}] \end{array} \right]$$

LFG glue uses the f-structure to constrain assembly by building references to it into the typed meanings that are to be assembled.

This is done by supplementing each literal with a reference to an ‘f-structure correspondent’. We’ll represent this by writing the type-literals as f-structure labels, subscripted with the semantic type. For (a) above, for example, the now-supplemented typed meanings might be:

¹⁰This is a linguistic version of the ‘no copying, no deletion’ principle of quantum logic, enlighteningly explored in Baez and Stay (2008). Luckily for linguists, glue seems to need only a minuscule fraction of the sophistication needed for physics.

- (19) *Bert* : g_e
Ernie : h_e
Like : $h_e \rightarrow g_e \rightarrow f_p$

If Matching is taken to require identity of f-structure labels as well as types, then there is only one way to put together the above typed meanings:¹¹

- (20)
-
- ```

graph TD
 fp["f_p^-"] --> ge["g_e^+"]
 fp --> ep["e -> p^-"]
 ge --> gem["g_e^-"]
 gem --- Bert["Bert"]
 ep --> he["h_e^+"]
 ep --> eep["e -> e -> p^-"]
 he --> hem["h_e^-"]
 hem --- Ernie["Ernie"]
 eep --- Likes["Likes"]

```

So, if we can get the syntax to control this supplementation of the literals, then the syntax will constrain the combination of the meanings. Fortunately, we can do this with the usual LFG technique of instantiation.

Suppose we include in lexical entries, alongside of the usual stuff, additional components called ‘meaning-constructors’, which will simply be typed meanings, with f-structure designators formed in the usual way with the  $\uparrow$ -arrow and GF names (and, sometimes,  $\downarrow$  as well). Entries for the words in our current examples would be:

- (21) *Bert* :  $\uparrow_e$   
*Ernie* :  $\uparrow_e$   
*Like* :  $(\uparrow \text{OBJ})_e \rightarrow (\uparrow \text{SUBJ})_e \rightarrow \uparrow_p$

To produce the f-structure (18a), we’d use these lexical entries of the words in this tree:

- (22)
- 
- ```

graph TD
    S --> NP1[NP]
    S --> VP[VP]
    NP1 --> N1[N]
    N1 --- Bert["Bert"]
    VP --> V[V]
    V --- likes["likes"]
    VP --> NP2[NP]
    NP2 --> N2[N]
    N2 --- Ernie["Ernie"]
  
```

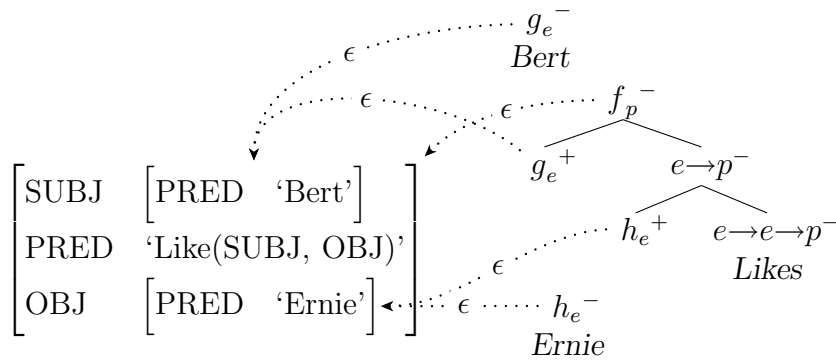
And then, as a consequence, the functional designators in the meaning-constructors will evaluate to yield the f-structure labels used in (19).

¹¹We are only writing in the f-structure component for the literals, to reduce redundancy.

A slightly different way to look at what is going on here, which leads to a useful pictorial representation, is this. We can think of the functional designators and f-structure labels as defining a correspondence relation between the semantic combination tree on the one hand, and the f-structure on the other. It has been customary since Kaplan (1987) to refer to correspondence between f-structure and semantics as ‘ σ ’, but since this one goes in the opposite direction from the f-structure-to-meaning direction that’s usually assumed, we shall here call it ϵ , for ‘expresses’ (I’ve used σ in some previous work, ϵ in Andrews (2008a)).

The regular LFG grammar will now generate an f-structure together with a collection of meaning-constructors, whose literals are connected to various parts of the f-structure by the ϵ relation:

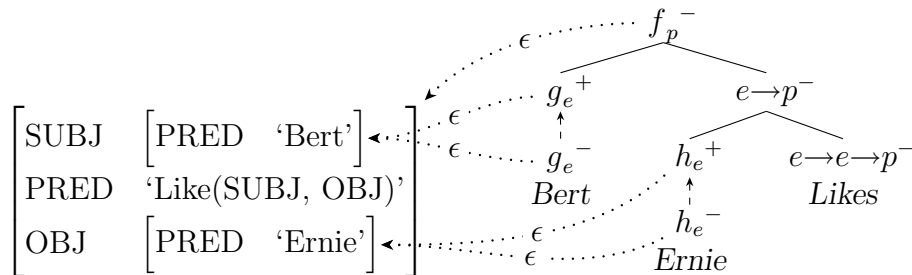
(23)



The constructors are at this point just floating around, attached to the f-structure by the ϵ -links, but not yet assembled into a coherent meaning.

The final step is to assemble them, so that all of the rules are satisfied, including the new one that axiom-linked nodes must be linked to the same piece of f-structure. For (23), there is only one way to do this:

(24)



Matching of the ϵ -correspondent means that if we contract the axiom-links, ϵ has a unique value for each merged literal, and so is a function, just like the ϕ -correspondence from phrase-structure to f-structure, but running in the opposite direction, from meaning to f-structure rather than overt form to f-structure.

It is hopefully evident that if the sentence had been *Ernie likes Bert*, then only the other sensible assembly would work, so we have achieved our goal of getting the syntax

to constrain the meaning. But, so far, we haven't used the glue system to represent anything that isn't fairly clearly represented in the f-structure. Ultimately, we will do this by considering quantifiers and scope, but, first, we need to make a detour into a somewhat logical topic, the 'lambda-calculus'. But before doing this, I will close this section by noting that Andrews (2007a, 2008b) describe an alternative technique for producing the meaning-constructors from the syntactic structure that is less reliant on the c-structure than the (standard) one used here.

4 Substitution and 'Lambda-calculus'

So far we've merely specified how meanings can be put together, based on the idea that some meanings are functions (or, more generally, 'function-like things', such as exponential objects in categories) that apply to others. However, we will also be wanting to say something about the 'substance' of some of these functions, which requires using more than just unanalysed names on the meaning-side (left side) of the constructors. One way to motivate this is to consider the problem of explaining the relationships between different uses of lexical items. It is clear that the intransitive and transitive use of *boil* in these examples have different but related meanings:

- (25) a. The water boiled
 b. John boiled the chicken

If we know what *boil* in (a) means, we can do a reasonable (although probably not complete) job of describing the meaning of (b) as follows:

- (26) [John put the chicken_{*i*} in boiling water for some time.]_{*j*}
 Because of this_{*j*}, after this_{*j*}, the chicken was not like it was before.
 John wanted it_{*i*} to be the way it was after this_{*j*}.

This is in a modified version of Wierzbicka's Natural Semantic Metalanguage, where we've used subscripts to indicate coreference, and assumed the availability of *put*, *boiling* and *water* as either semantic primitives (rather unlikely) or previously defined concepts ('molecules'). To get a general sense of transitive *boil* that applies to leather, eggs, etc, it is standard practice to formulate an 'explication' like this:

- (27) X boiled Y
 X put Y in boiling water for some time
 Because of this, after this, Y was not like Y was before this.

Here I've left out the brackets and subscripts, because the 'variables' X and Y seem to suffice for the disambiguations.

So then the idea of substitution is that to get a meaning for a particular sentence, you need to plug in 'real meanings' for the X and Y positions. Some general discussion of what is involved in doing this for NSM can be found in Andrews (2006).

The standard tool used in formal semantics for describing substitutions of this sort is the ‘lambda-calculus’, which can be regarded as a technique for supplementing a formalized language with two additional constructions, ‘abstraction’, and ‘application’. An application consists of a ‘function’ and an ‘argument’, usually written in that order, with the argument in parentheses:

$$(28) \quad \begin{array}{c} f(a) \\ \swarrow \quad \searrow \\ \text{function} \quad \text{argument} \end{array}$$

The negative polarity implications in our glue-trees are in fact just syntax trees for applications, with the function written on the right rather than the left, for convenience of the layout for line-drawing.

An abstraction on the other hand consists a ‘head’ and a ‘body’, where the head consists of the symbol λ and a variable, and the body is a formula in which the variable may appear any number of times, including 0:

$$(29) \quad \begin{array}{c} \lambda x.b \\ \swarrow \quad \searrow \\ \text{head} \quad \text{body} \end{array}$$

The variable in the head is something that can also appear in the body, in positions where something else is ultimately meant to be substituted in.

We don’t yet have abstraction in our glue-structures, but this will change.

The basic idea of substitution is that it’s stupid (under some circumstances) to write something like this:

$$(30) \quad (\lambda x.b)(a)$$

where we’ve created an abstraction with body b and variable x , and applied it to argument a . Rather, this combination of the two constructions should be replaced by a simpler format in which neither appears, but, instead, a replaces all ‘free’ occurrences of x in b , ‘free’ being a technical concept that we’ll have a bit to say about later. A very simple example is that $(\lambda x.Yell(x))(Bert)$ should be replaced by $Yell(Bert)$ (people often put parentheses around the lambda-abstraction, to help clarify what is supposed to apply to what, even when it’s not really necessary). For a more complicated example, work out what $(\lambda x.Not(Yell(x)))(Bert)$ should be replaced by (answer in (102)).

To define this more precisely, we use a rule called ‘ β -reduction’, which, in words, says that the result of applying a lambda-abstraction to an argument in the application construction is the same thing as the body of the abstraction, with the argument substituted for all ‘free’ occurrences of the variable that appears in the body of the abstraction. In symbols, one writes:

$$(31) (\lambda x.b)(a) \Rightarrow_{\beta} [a/x]b$$

The idea of the $[a/x]$ notation is that one ‘divides out’ x and ‘multiplies in’ b . Textbooks such as Hindley and Seldin (1986) provide a rigorous definition of this notation, but I don’t think we need that here.

So what makes an occurrence of a variable x ‘free’? Nontechnically, it must sit neither in a head, nor in a body whose head is ‘ λx ’. So, for example, in an expression such as $(\lambda x.Dog(x))(x)$, only the third and last occurrence of x is free. A good technique for avoiding problems with free versus non-free is to use a different variable for each λ in all of the abstractions that are going to be used in a discussion.

Finally, we need to connect lambda-calculus to types. There are two general kinds of lambda-calculus, ‘typed’, and ‘untyped’. The latter doesn’t have types, and is therefore a bit simpler to formulate, but its meaning and behavior can be rather intimidating. In the typed lambda calculus, every expression has a type, and there is an implicational type constructor, which is connected to the application and abstraction constructions as follows:

- (32) a. If f is of type $A \rightarrow B$, and a is of type A , then $f(a)$ is of type B .
 b. If x is of type A , and b is of type B , then $\lambda x.b$ is of type $A \rightarrow B$.

The typing rule for (a) is clearly the same as the one we’ve used for our negative-polarity implications, which are, so far, the only ones we have. Indeed we can say that these are the applicational construction in our system of glue-formulas. We don’t have abstraction in these structures yet, but this will change.

But we can still use abstraction to provide some integration of our compositional semantics with the lexical semantics that we’ve been suggesting. The idea is to convert the glue-structures into application structures in the formal language that we use for presenting meanings, and then use β -reduction to implement the substitutions.

A convenient way to do this is with a ‘semantic labelling’ procedure whereby lambda-terms in the meaning language are assigned to glue-tree nodes in a bottom-up manner, following the dynamic graph. So far, the leaves of the dynamic graph will be only be rightmost daughters with semantic labels, so the following rule will suffice:

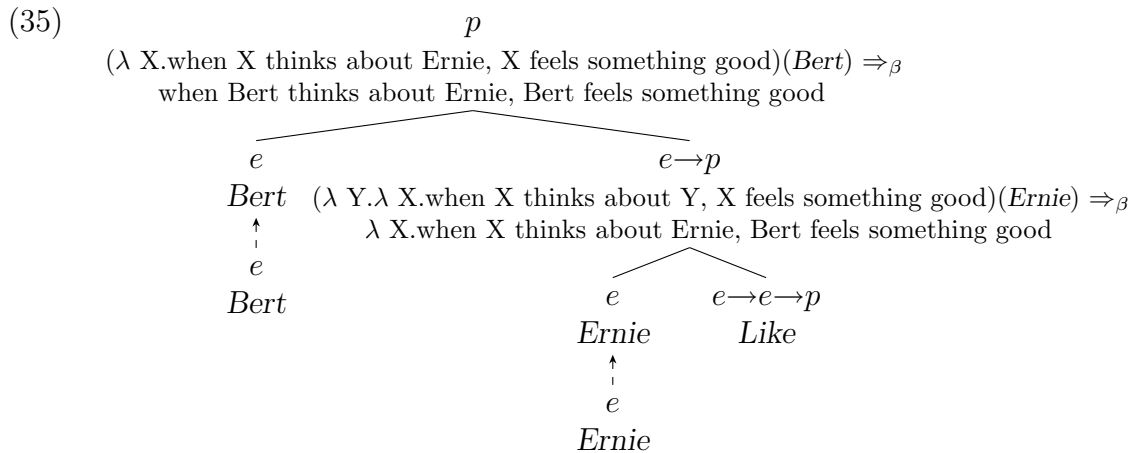
- (33) a. The semantic label of a negative implication is that of its right daughter applied to that of its left daughter.
 b. The semantic label of the target of an axiom-link is that of its source.

This is a completely trivial operation, but serves the function of setting things up notationally for the β -reduction rule,

For example, suppose we’ve decided that the meaning of *Like* is:

(34) when X thinks about Y, X feels something good

Then we could produce an explication for *Bert likes Ernie* as follows, taking some notational shortcuts to fit things onto the page:



Actually doing substitutions into a natural language NSM entails a variety of problems of agreement, case-marking, etc, briefly discussed in Andrews (2006), but hopefully the basic idea is viable.

It should also be clear that we only have to do the semantic labelling and substitutions if we want to make use of the information provided by the definitions of the lexical items: if all we're interested in is how the lexical meanings are to be combined, the glue tree itself provides all the relevant information.

5 Quantifiers

Many ambiguities are reflected in f-structure in a fairly obvious way, but the 'scope ambiguities' in sentences such as these are not:

- (36) a. Everybody doesn't yell
 b. Everybody loves somebody

In LFG, the f-structures for these sentences would be, respectively:

- (37) a.
$$\left[\begin{array}{l}
 \text{SUBJ} \quad \left[\text{PRED} \quad \text{'Everybody'} \right] \\
 \text{POLARITY} \quad \text{NEG} \\
 \text{PRED} \quad \text{'Yell(SUBJ)'}
 \end{array} \right]$$

- b.
$$\left[\begin{array}{l} \text{SUBJ} \quad \left[\text{PRED} \quad \text{'Everybody'} \right] \\ \text{PRED} \quad \text{'Love(SUBJ, SUBJ)'} \\ \text{OBJ} \quad \left[\text{PRED} \quad \text{'Somebody'} \right] \end{array} \right]$$

Each structure represents both meanings, and a satisfactory way to represent scope phenomena in f-structure itself has not yet been devised.¹²

Although it would not be crazy to suggest that scope is handled in a rather different fashion from the aspects of linguistic structure that are encoded in f-structure, there are nonetheless various phenomena that depend on scope. For example, the indefinite pronouns *somebody* and *anybody* differ that *somebody* resists being within the scope of negative, while *anybody* demands to be in the scope of a negative or certain other kinds of operators. Therefore, in these examples, (a) prefers wide scope for its quantifier object NP, while (b) requires narrow:

- (38) a. John doesn't like somebody
 b. John doesn't like anybody

Glue normally uses the 'generalized quantifier' analysis of Barwise and Cooper (1981), which can in fact be seen as a somewhat upgraded analysis of the treatment of quantifiers in Classical Antiquity. The basic idea is that a quantificational NP such as *everybody*, *somebody* or *nobody* is a 'predicate of properties', that is, something that is true or false of things of type $e \rightarrow p$, that is, something that is of type $(e \rightarrow p) \rightarrow p$. *Nobody*, for example, is true of a property *Yell* just in case the intersection of the collection of yellors with the collection of people is empty, while *Everybody* is true of *Yell* just in case the people are a sub-collection of the yellors (you might have to think about this for a bit to see it).

A typed meaning for *everybody* will therefore be:

- (39) *Everybody* : $(e \rightarrow p) \rightarrow p$

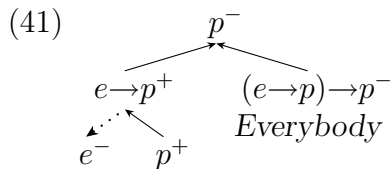
Converting this into our tree-format with the rules of (7), and assigning polarities, we get the structure:

- (40)
$$\begin{array}{c} p^- \\ \swarrow \quad \searrow \\ e \rightarrow p^+ \quad (e \rightarrow p) \rightarrow p^- \\ \quad \quad \quad \textit{Everybody} \end{array}$$

¹²Andrews and Manning (1993) might be the most conscientious attempt.

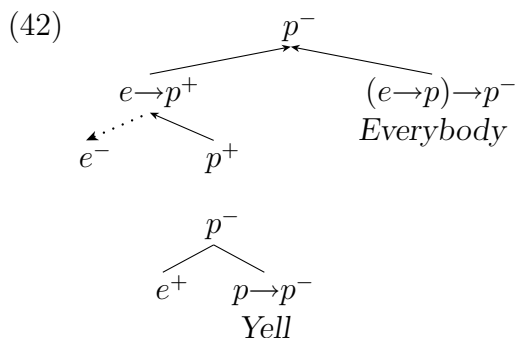
But now we're stuck, because we don't know what to do with the implicational left-daughter (so far, all left daughters have been literals). What we'll do is simply give the answer, then discuss it.

In the first place, we'll trivially reformulate our rules so that they build the tree and assign polarities at the same time. So, in the process of tree-building, the implicational left-daughter of (40) will get positive polarity. Then we decree that a positive implication will expand (downward) to a negative 'left pseudo daughter', labelled with the antecedent of the implication, and a positive 'right daughter', labelled with the consequent. The left daughter is described as 'pseudo' because the link that connects it to its mother is *not* in the dynamic graph, while the link that connects the right daughter to the mother is in the dynamic graph, oriented upwards. We represent this difference by connecting the left pseudo-daughter to its mother with a dotted line (oriented here from mother to daughter). So then (40) becomes (41), where we've also indicated the dynamic graph directionality of the solid links:

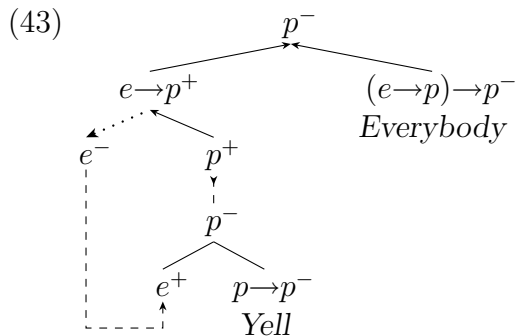


For a final version of the prefab-building rules, plus discussion of some more complicated cases, see Andrews (2008a). So how do we combine these new structures with other things?

We'll first look at the simplest case, for which the machinery is a bit cumbersome, but in ways that will be essential a bit later. Since *Everybody* is of type $(e \rightarrow p) \rightarrow p$, we should expect it to apply to something of type $e \rightarrow p$, such as, for example, *Yell*. So we have these two trees, suggestively arranged:

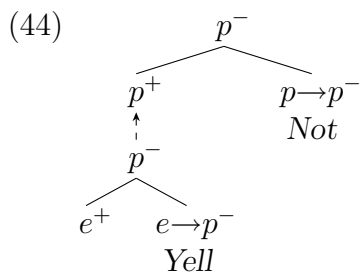


Now, the assembly rules say that we're supposed to run axiom links from positives to negatives of matching type, subject to the Correctness Criterion, and the only way to do this for (42) is like this:

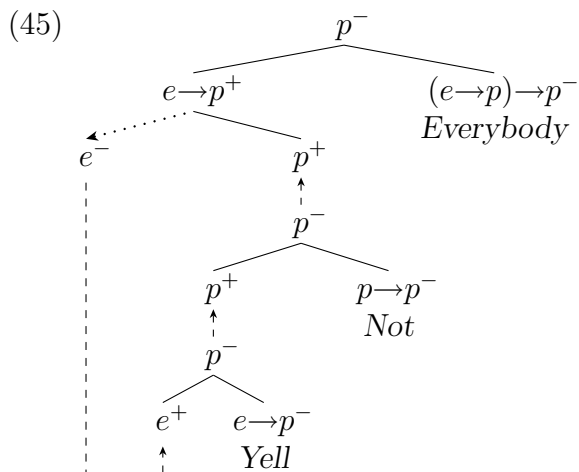


This could seem a bit excessive, because we're putting in two axiom-links to express one predicate-argument relation, namely, that *Yell* is the argument of *Everybody*

But for a more complicated example where this technique pays off, suppose we first put together typed meanings for *not* and *yell*, like this:



Then we can combine this with the *Everybody* constructor like this:

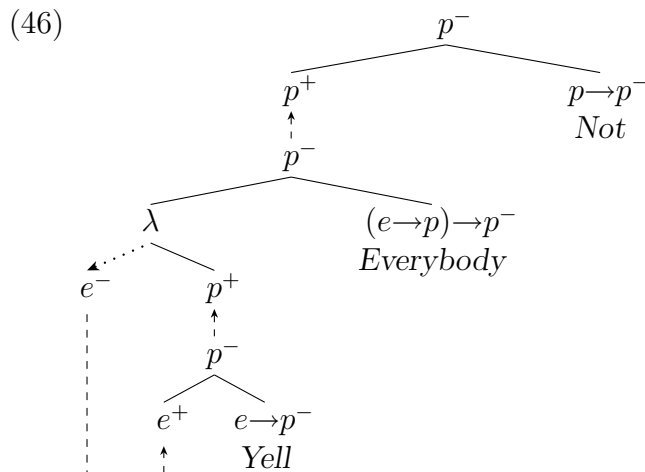


The intended sense of this is that everybody has the property of not yelling (wide-scope reading of the quantifier).

To motivate this intuitively, think of the negative e^- of the quantificational pronoun as shooting out 'test entities', which enter the sub-assembly (44), which returns 'true'

of ‘false’ (perhaps relativized to a ‘possible world’ or some other kind of context) to its top, un-axiom-linked negative p . If, whenever the test entity is a person, the returned value is ‘true’, then, the quantificational assembly (45) represents a true proposition, otherwise, a false one.

But there is another way of putting the pieces together, which delivers the narrow scope reading. For this, first combine the quantifier directly with the verb, as we did in the simple example, and then apply the negative to the result of doing that. But since the positive implications have the same type-assignment rule as lambda-abstractions, and their function is essentially the same, it is convenient to label them simply with λ , giving this for the narrow-scope reading:



This corresponds to shooting the test entities into the *Yell* predicate, and then applying negation to the result of the quantification.

Once the positive polarity implications are taken on board, it turns out that we need an additional clause to the Correctness Criterion:

- (47) Correctness Criterion (clause 2): From every left-pseudo-daughter of an implication, the dynamic graph must provide a path to the right-daughter (and, therefore, equivalently, to the implication itself).

As a relatively difficult exercise, you might try to devise a structure that satisfies the principles so far except for (47), but doesn’t make sense (one is provided in Andrews (2008a)).

Another perspective emerges from the ‘semantic readings’ of the previous section. Our representations of the quantifier meanings are rather ‘austere’ compared to what one would find in any logic or formal semantics course. In introductory logic, for example, a quantified NP such as *everybody* would probably be read as a combination of the ‘universal quantifier’ \forall , and the ‘material implication’ \supset , which obeys principles making it roughly equivalent to the ‘*if...then*’ construction. In formal semantics, on the other

hand, it is common to find quantifiers in a ‘tripartite’ construction where they relate a variable and two propositions. These possibilities are illustrated here, where Φ is the formula which is supposed to be true of everybody:

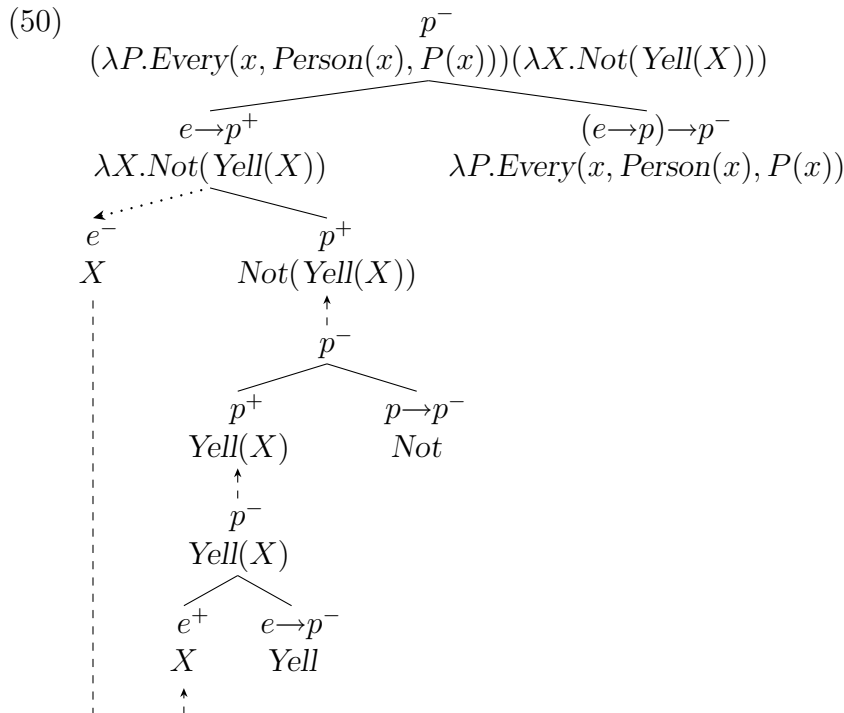
- (48) a. $(\forall x)(Person(x) \supset \Phi)$
 b. $Every(x, Person(x), \Phi)$

We can use either of these decompositions in a semantic reading method by explaining what to do with the positive implications.

What we do is add the following two principles:

- (49) a. the semantic reading of a left pseudo-daughter is a new variable.
 b. the semantic reading of a positive implication is the lambda-term with the variable assigned to its left pseudo-daughter in its head, and the reading of its right daughter as its body.

Given these rules, the semantic reading for (45) can be calculated as follows:



Working out the final reading of the top p^- is a bit of a chore, because it requires two β -reductions:

$$\begin{aligned}
(51) \quad & (\lambda P. \text{Every}(x, \text{Person}(x), P(x)))(\lambda X. \text{Not}(\text{Yell}(X))) \Rightarrow_{\beta} \\
& \text{Every}(x, \text{Person}(x), (\lambda X. \text{Not}(\text{Yell}(X)))(x)) \Rightarrow_{\beta} \\
& \text{Every}(x, \text{Person}(x), \text{Not}(\text{Yell}(x)))
\end{aligned}$$

As with the semantic readings for negative implications, those for positive ones aren't needed until we're doing something with the internal structure of the meanings of the constructors, since if were not, the glue trees themselves are a perfectly adequate representation of the semantic composition.

So next, we move on to the problem of connecting quantifiers to the grammatical structure. Here the issue is that the propositional literals connect to some higher structure. Often this is the clause immediately containing the quantifier NP, but not always:

(52) John thinks that Mary likes everybody

This sentence has a (less natural) reading where, for every person x , John thinks that Mary likes x .

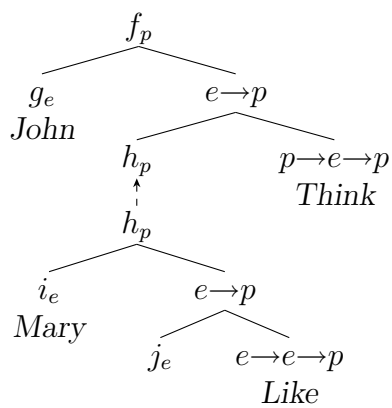
The treatment of this that I will use here is based on the 'propositional glue' system of Andrews (2008a), which is mathematically simpler than the ones used in the previous glue literature. The basic idea is to use local names and iofu paths to connect the type p literals to the f-structure:

$$\begin{aligned}
(53) \quad & \text{Everybody} : (\uparrow_{\epsilon} \multimap \%H_p) \multimap \%H_p \\
& (\text{GF}^* \uparrow) = \%H
\end{aligned}$$

The 'local name' (Dalrymple 2001:146-148) $\%H$ can represent any piece of the f-structure which, as stated by the iofu equation (Dalrymple 2001:145) underneath, can be gotten to by following an upward path of grammatical functions from \uparrow .

To see what (53) does, first consider what (52) gets as functional and glue structure without the quantificational pronoun (omitting the polarities, since supplying them should be routine by now, and representing ϵ with f-structure labels in the glue-structure):

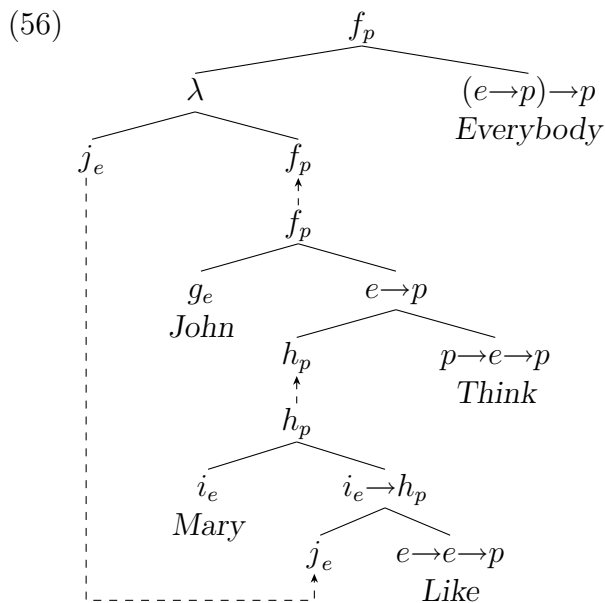
$$(54) \left[\begin{array}{l} \text{SUBJ } g: [\text{PRED 'John'}] \\ \text{PRED 'Think(SUBJ, OBJ)'} \\ f: \left[\begin{array}{l} \text{SUBJ } i: [\text{PRED 'Mary'}] \\ \text{COMP } h: \left[\begin{array}{l} \text{PRED 'Like(SUBJ, OBJ)'} \\ \text{OBJ } j: [\text{PRED 'Everybody'}] \end{array} \right] \end{array} \right] \end{array} \right]$$



By what we've said so far, the tree-form of the quantifier constructor will be:

$$(55) \begin{array}{c} \%H_p \\ \lambda \quad (e \rightarrow p) \rightarrow p \\ j_e \quad \%H_p \quad \text{Everybody} \end{array}$$

j_e here will have to be axiom-linked to j_e in (54), but for the $\%H$ -literals, things are more complicated, since $\%H$ can be instantiated to either f or h . Either of these choices provides a sensible reading, the former wide-scope, the latter narrow. Here is the former:

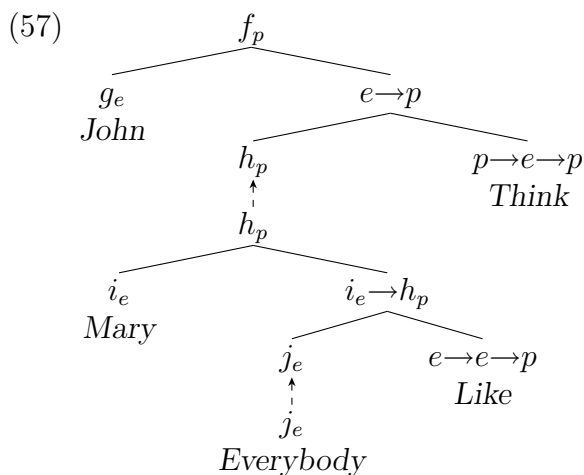


As an exercise, write out the structure for the narrow-scope reading.

In this example, the scope ambiguity is produced by two different instantiations of the propositional literal, but in the ambiguity of *everybody loves somebody*, there is only one possible instantiation for this literal. But there are two possible arrangements of the quantifiers, leading to two readings.

A final point about the quantifiers that is worth considering is this. The ambiguities they induce can be rather subtle and hard to see, so it would not be crazy to suggest that they arise at a relatively late level of linguistic processing that is not in fact always performed. The present treatment is actually consistent with this, thanks to some observations by Gupta and Lamping (1998).

Suppose that, in a preliminary assembly, we simply ignore the propositional literals, and pretend that they're not there. The quantifier will then act, for purposes of assembly, as if it was merely an expression of type e , and will fit into the tree of (54) like this:



This provides an ‘underspecified’ representation for both readings, and one of Gupta and Lamping’s points is that if we need a more specific representation, we can derive it from (57) or equivalent by choosing some f-structure correspondent for the propositional literals, and then ‘splicing in’ the *Everybody* constructor to the appropriate position. There are also various clever tricks one can use to record the viable possibilities, but going into that is more a topic in Computational Linguistics than basic glue. This opens up some interesting connections between glue and Minimal Recursion Semantics (Copestake et al. 2005).

6 Common Nouns and the Meaning-side

The somewhat arduous work we’ve done with β -reduction on the meaning-side begins to pay off when we look at how common nouns fit in with quantifiers. We need to account for the relationships between quantificational pronouns such as *everybody*, and quantified full NPs such as *every dog*, which we can now do, essentially by lambda-abstracting on two property-variables rather than just one.

We start by considering NPs such as *every dog*. It is a fairly standard analysis of common nouns that they are of type $e \rightarrow p$, meaning that an entity might be, or not be, a dog. So, in a sentence such as:

(58) Every dog barks

the effect of the quantificational determiner seems to be to compare the collection of entities that are dogs with the collection of entities that are barkers, and return ‘true’ if the former is included in the latter, ‘false’ otherwise. Other quantificational determiners, such as *some* and *no*, seem to effect similar comparisons of the ‘extensions’ of properties, that is, the collections of thing that those properties are true or false of. *some* for example, appears to require that the extensions overlap, *no* that they don’t.

The semantic type of these quantificational determiners therefore seems to be:

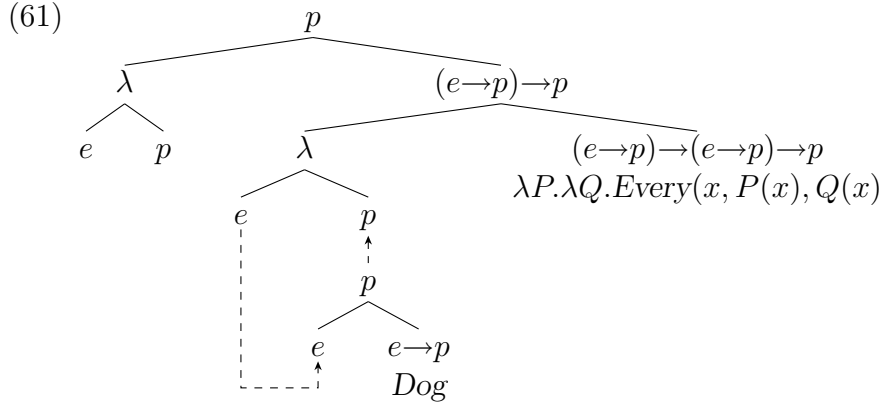
(59) $(e \rightarrow p) \rightarrow (e \rightarrow p) \rightarrow p$

This would tend to be intimidating to people without a lot of background, so we’ll walk through it. According to what we’ve been saying, the meanings of quantificational determiners appear to be relations between properties (indeed, ones that depend only on the extensions, which is really quite interesting). So by the ‘Currying’ technique we’ve been using, their type would be something like **property** \rightarrow **property** $\rightarrow p$. But ‘properties’ are of type $e \rightarrow p$, so this is ‘just’ (59).

Next, it is plausible to associate the first of these properties with the nominal, the second with the sentence in which the common noun is used (on the general grounds of applying the ‘closest’ thing first). Given our previous work, this will be the appropriate internally-structured meaning-side:

(60) $\lambda P.\lambda Q.Every(x, P(x), Q(x))$

Now suppose this appears in a structure like this:



The semantic reading for the innermost λ -node is $\lambda X.Dog(X)$ (we consider equivalent all variants of this which differ only in their choice of novel variable for X). This ought to mean just the same thing as *Dog*, but β -reduction won't effect this identification. There is another principle, called η -reduction, which we'll see later, that will, but we don't really need it, because when we apply this as argument to the *every* meaning, we get:

(62) $(\lambda P.\lambda Q.Every(x, P(x), Q(x)))(\lambda X.Dog(X)) \Rightarrow_{\beta}$
 $\lambda Q.Every(x, (\lambda X.Dog(X))(x), Q(x)) \Rightarrow_{\beta}$
 $\lambda Q.Every(x, Dog(x), Q(x))$

Now, *every dog* has assembled to produce something with the same potential for further assembly *everybody*, so that the treatment of sentences such as *every dog doesn't bark* will follow what we've already done.

7 Intersective Adjectives

We are now set up to treat 'intersective' adjectives such as *crazy*, as might be found in an example such as *every crazy dog barks*. The idea of intersective adjectives is that they apply as it were jointly to the referent of the NP along with the head nominal. So something is a 'crazy dog' if and only if it is a dog, and also crazy. Some non-intersective adjectives are *putative* and *self-styled*:

- (63) a. A putative goose is in the backyard.
 b. A self-styled physicist announced a water-burning engine.

There's no such thing as just being 'putative', and a putative goose might not actually be a goose (but, perhaps, merely a duck). Similarly for a self-styled physicist.

So how to accomplish this intersective interpretation. For reasons presented in Dalrymple (2001:ch.10), it seems to be necessary to use two meaning-constructors. The first is associated with the lexical item and simply expresses a property:

$$(64) \text{ Crazy} : \uparrow_e \rightarrow \uparrow_p$$

Some motivation for this constructor is that it is very close to what we want for the predicative use of these adjectives. If for example we follow the usual LFG analysis of treating predicate adjectives as APs bearing the XCOMP grammatical function, all we need is the following variation of (64):

$$(65) \text{ Crazy} : (\uparrow \text{SUBJ})_e \rightarrow \uparrow_p$$

Adjectives such as *asleep* that can occur as predicate adjectives but not attributives will have entries with the glue-side of (65) but not (64).

But now, if we just introduce an intersective attributive adjective under an AP introduced with the ADJUNCT GF, as is generally assumed, \uparrow in (64) will then instantiate to the f-structure correspondent of this AP, producing the following f-structure and instantiated constructors:

$$(66) \left[\begin{array}{l} \text{PRED} \quad \text{'Dog'} \\ \text{ADJUNCTS} \quad \left\{ g: \left[\text{PRED} \quad \text{'Crazy'} \right] \right\} \end{array} \right]$$

$$\text{Dog} : f_e \rightarrow f_p$$

$$\text{Crazy} : g_e \rightarrow g_p$$

We need a further constructor to put all this together, which can be plausibly associated with the PS rules (although Dalrymple includes it in the lexical entry of the adjective, as an additional constructor).

The idea is that the head N is already providing one property, the AP another, and what this additional constructor is doing is combining them with logical ‘and’, often symbolized as \wedge (among various other possibilities). If we suppose that these adjectives are introduced as AP under \bar{N} recursively, this rule will do the job:

$$(67) \bar{N} \quad \rightarrow \quad \text{AP} \quad \bar{N}$$

$$\downarrow \in (\uparrow \text{ADJUNCTS}) \quad \uparrow = \downarrow$$

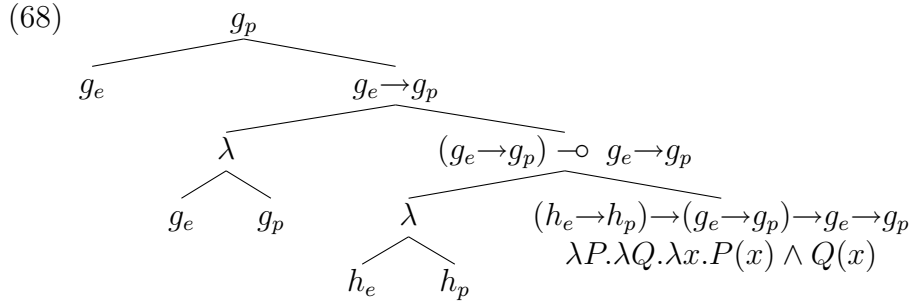
intersective

intersective:

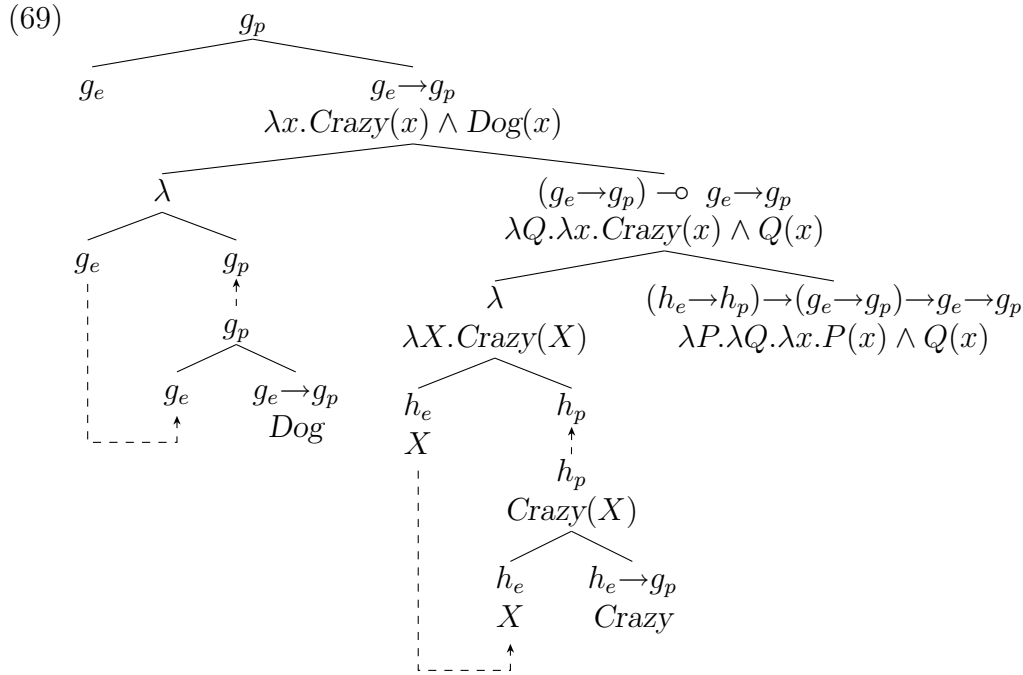
$$\lambda P. \lambda Q. \lambda x. P(x) \wedge Q(x) : (\downarrow_e \rightarrow \downarrow_p) \rightarrow (\uparrow_e \rightarrow \uparrow_p) \rightarrow \uparrow_e \rightarrow \uparrow_p$$

A walkthrough of how this rather intimidating-looking thing works might be useful.

Expanding **intersective**, with some typical instantiations, to a tree, we get:



Here, we've put the f-structure information into the complex node labels, as well, to make the construction easier to follow. Combining this with the *crazy* and *dog* constructors, we get:



β -reduced semantic readings are provided for some but not all of the nodes. You can supply those for the upper λ subassembly as an exercise.

Writing out these tree structures can be a bit of a chore, but when you understand what is going on, you can abbreviate things considerably by first writing out the instantiated constructors in the standard format:

(70)

$$\begin{aligned} \text{Dog} &: f_e \rightarrow f_t \\ \text{Crazy} &: g_e \rightarrow g_t \\ \lambda P . \lambda Q . \lambda x . P(x) \wedge Q(x) &: (g_e \rightarrow g_t) \rightarrow (f_e \rightarrow f_t) \rightarrow f_e \rightarrow f_t \end{aligned}$$

and then drawing in axiom-links:

$$\begin{array}{l}
(71) \quad \text{Dog} : f_e \rightarrow f_t \\
\text{Crazy} : g_e \rightarrow g_t \\
\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x) : (g_e \rightarrow g_t) \rightarrow (f_e \rightarrow f_t) \rightarrow f_e \rightarrow f_t
\end{array}$$

Two problems with this method are (a) there isn't any convenient place to write in the semantic readings (b) one can make mistakes in checking the Correctness Criterion. This latter, more serious, problem can be addressed by drawing in the Dynamic Graph connections between the literals as solid lines:

$$\begin{array}{l}
(72) \quad \text{Dog} : f_e \rightarrow f_t \\
\text{Crazy} : g_e \rightarrow g_t \\
\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x) : (g_e \rightarrow g_t) \rightarrow (f_e \rightarrow f_t) \rightarrow f_e \rightarrow f_t
\end{array}$$

One can then check the Correctness Criterion by tracing the paths. For clause 2, trace the ones from the negative antecedents, and make sure they go through the corresponding consequents. For clause 1, check that the paths from the Final Outputs of the individual constructors go to the Final Output of the whole assembly. To perform this part of the check here, one would have to provide more constructors, such as for a quantificational pronoun and intransitive verb.

The attributive constructor is complex, but similar in form to the one used for apposition in Australian language NPs by Sadler and Nordlinger (2008), so working through this should set you up to manage that.

8 Adverbial Modification

An important motivation for the two-constructor treatment of intersective adjectives is to account for the possibility of modification by 'propositional' adverbs such as *obviously*:

$$(73) \text{ an obviously crazy dog was barking}$$

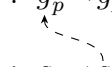
Exactly what the phrase-structure of such modifiers are is hard to work out, but I will assume this rule:

$$\begin{array}{l}
(74) \text{ AdjP} \rightarrow \quad \text{AdvP} \quad \text{AdjP} \\
\quad \quad \quad \downarrow \in (\uparrow \text{ADJUNCTS}) \quad \uparrow = \downarrow
\end{array}$$

Now, what the adjunct-members are supposed to do is apply to the predication produced by the adjective they apply to. This can be accommodated by using some iofu in the meaning-constructor for these adverbs, as follows:

$$(75) \quad \text{Obvious} : \%F_p \rightarrow \%F_p \\ \%F = (\text{ADJUNCT} \in \uparrow)$$

Now the instantiated constructors for *obviously crazy*, with some assembly, will be:

$$(76) \quad \text{Obvious} : g_p \rightarrow g_p \\ \text{Crazy} : g_e \rightarrow g_p$$


You might be able to work out that the ultimate result is that this AP will make the contribution $\lambda x. \text{Obvious}(\text{Crazy}(x))$ to the meaning of the NP, which is what is wanted.

This constructor will furthermore produce the right results for modification of sentences as well as intersective adjectives. so that for example from the f-structure of *every dog obviously barked*, we can get two glue-structures. There are plenty of more issues to consider with a glue analysis of adverbs, but this is enough for an introduction.

9 Notational Interlude

Conceptually, we've gone through enough material to actually read a glue paper, but there are still some notational issues to deal with.

The first is that most, but not all, glue presentations use the symbol \multimap , colloquially read 'lollipop', instead of \rightarrow , for implication. When this symbol appears, one can assume that the logic being used is 'linear', instead of 'classical', as will be explained in the next section. But this really has nothing to do with the nature of implication *per se*, but rather with the general environment in which it is working, so that the use of this symbol (which comes from category theory), could be regarded as unnecessarily intimidating.

The second issue is the 'semantic projection'. We've been supposing that the ϵ -correspondents of glue literals are f-structures, but it is normally assumed that they are residents of a special 'semantic projection' σ , that comes off f-structure, and has a rather small number of attributes used for semantic bookkeeping. Indeed, it seems to me, hardly anything at all, so I propose dispensing with this level in Andrews (2008a), and various previous discussions.¹³

The semantic projection is constructed as usual by the meaning-constructors, so that for example (67) becomes (also using t for p):

$$(77) \quad \bar{N} \quad \rightarrow \quad \text{AP} \quad \bar{N} \\ \downarrow \in (\uparrow \text{ADJUNCTS}) \quad \uparrow = \downarrow \\ \textbf{intersective} \\ \textbf{intersective:}$$

¹³See also Kokkonidis (2008).

$$\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x) : ((\downarrow_{\sigma} \text{VAR})_e \multimap \downarrow_{\sigma t}) \multimap ((\uparrow_{\sigma} \text{VAR})_e \multimap (\uparrow_{\sigma} \text{RESTR})_t) \multimap (\uparrow_{\sigma} \text{VAR})_e \multimap (\uparrow_{\sigma} \text{RESTR})_t$$

Given a \bar{N} such as *big dog*, this PS rule will produce the following combination of f-structure and semantic projection:

$$(78) \left[\begin{array}{cc} \text{PRED} & \text{'Dog'} \\ \text{ADJUNCTS} & \left[\begin{array}{cc} \text{PRED} & \text{'Big'} \end{array} \right] \end{array} \right] \xrightarrow{\sigma} \left[\begin{array}{c} \text{VAR} [\] \\ \text{RESTR} [\] \end{array} \right] \\ \xrightarrow{\sigma} \left[\text{VAR} [\] \right]$$

The ϵ -correspondents of the glue literals will then be occupants of this new projection, rather than f-structures.

Notice that the semantic projection is quite disconnected, and could be viewed as being merely a place to park certain attributes such as VAR and RESTR, which aren't traditionally found in f-structure, and don't have the same kind of motivation as the things that are. Indeed, there is a bit of a question as to what the motivation for these attributes is in the first place.

One possible motivation is that it might be bad to have glue literals of different types, such as e and p , connected to the same syntactic object, such as the f-structure of the subject. So literals of type e are connected to the value of a VAR attribute on the semantic projection, while those of type t (for which we've been using p , are connected to either a semantic projection itself or to a RESTR (for 'restriction') attribute on this projection.

However, so far at least, there is hardly any solid argument for this (an argument presented by Asudeh (2005b) is countered in Andrews (2008a)), and it seems to be mostly esthetics and some traditions. So here, the semantic projection will be dispensed with. But, given a good argument for it, there would be no problem in re-introducing it into the present approach, although the structure diagrams would become more complex, with the (rather sparsely populated and disconnected) semantic projection objects sitting between the f-structures and the glue trees.

A final point is that in the standard presentation, the ϵ projection is entirely implicit in the co-labelling between glue-proof literals and semantic projection structures, although its existence is sometimes exploited, for example in Crouch and van Genabith (1999) and Asudeh and Crouch (2002).

We can now read and use constructors, except for analyses that use 'tensors' (mainly, anaphora), but the way in which we are using them is still different from the standard, since most of the glue literature uses deductions, and our format for proof-nets is non-standard. So deductions come next on the list of topics.

10 Glue by Natural Deduction

There have unfortunately been two schemes of deduction used in glue. The older literature used a conceptually immensely important¹⁴ but very unintuitive system called ‘Gentzen Sequents’, which we’ll ignore here (see Crouch and van Genabith (2000) when you want to know more about them), but then mostly shifted over to the ‘tree style Natural Deduction’ format, used for example by Dalrymple and Asudeh. To which we now turn.

Superficially, the Natural Deduction formulation is quite different from our present prefab tree format, but they are really the same thing, except for the not-so-minor point that the prefab tree format represents only proofs in ‘normal form’. Which is standardly discussed with respect to the Natural Deduction format, so here we go.

Natural deduction is a family of similar techniques of formal logical deduction developed between the 1930’s and the 1960’s to provide a more intuitive approach than the ‘Hilbert systems’ that were first developed to formalize modern logic (see Pelletier (1999) for a historical discussion). The basic idea was to formalize the kinds of thought-processes actually used in normal mathematical arguments, which are quite different from the structure of proofs in a Hilbert system. Here we will be concerned with only one form of Natural Deduction, Gerhard Gentzen’s ‘tree style’, and only one connective, implication, which we’ll write as \rightarrow .

Natural Deduction works by usually providing, for each connective, a pair of rules, one to eliminate that connective, another to introduce it. Implication elimination is the classic rule of ‘Modus Ponens’, whereby, it is held that if one assents to ‘if A then B’, and also to ‘A’, one ought to assent to ‘B’ (if you don’t, you might be deemed not worth trying to have a sensible discussion with). We can see why it’s called elimination of implication, since one of the two premises, the so-called ‘major premise’ has an implication, which is gone in the conclusion. In tree-style, the rule would typically be written like this:

$$(79) \frac{A \rightarrow B \quad A}{B} \mathcal{E} \rightarrow$$

The premises appear above the line, the conclusion, which is supposed to follow from them, below it, and the name of the rule justifying this reasoning step appears to the side, being \mathcal{E} for eliminations, \mathcal{I} for introductions, followed by the connective being managed (needless to say, this is one of many notations for the justifications). There seems to be a tradition of putting the major premise on the left, though it doesn’t really matter.

ND deductions are produced by combining such rule applications, using the conclusion of one application as one of the premises of another. Below, for example, we derive C

¹⁴See for example Melliès (2008), the earlier sections of which are quite accessible to people with a smattering of logic, although the mathematical level rises inexorably.

from A , B , and $A \rightarrow B \rightarrow C$:

$$(80) \frac{\frac{A \rightarrow B \rightarrow C \quad A}{B \rightarrow C} \mathcal{E} \rightarrow \quad B}{C} \mathcal{E} \rightarrow$$

At this point you might have noticed that these deductions look very much like glue-assemblies without lambdas, with the axiom-links contracted, but put rightside-up, on their roots, rather than hung upside-down, as linguists tend to like their trees.

This resemblance becomes stronger when we introduce the introduction rule. The idea behind this rule is that if we can derive B from the assumption A and perhaps some others, then we can derive $A \rightarrow B$ from those other assumptions alone, without A . So, for example, if we can prove ‘the world is doomed’ from the premises ‘Bush is a moron’ and ‘Ahmedinejad is a maniac’, then we can prove ‘if Bush is a moron, then the world is doomed’ from the premise ‘Ahmedinejad is a maniac’. The rule is formulated like this:

$$(81) \frac{\begin{array}{c} [A]^i \\ \vdots \\ B \end{array}}{A \rightarrow B} \mathcal{I} \rightarrow^i$$

It is implicit in this format that there may be other assumptions involved in the justification for B , and the superscripted brackets indicate that the assumption so-bracketed is ‘discharged’, that is, no longer needed to derive the conclusion.

This formulation is temporally telescoped, since it should be regarded as the output resulting from a prior stage of the derivation that looks like this:

$$(82) \begin{array}{c} A \\ \vdots \\ B \end{array}$$

But no information is lost by the notation, due to the superscripting of the introduction-step with the premise that it discharges. For the portion of the proof-tree that is above the discharge step will depend on the premise (i.e. the brackets and superscript are to be ignored), while the portion below does not (the brackets are noted as indicating that the premise is no longer needed to derive the current conclusion).

With both the introduction and elimination rule, we can begin to prove a wider variety of results. A very simple proof is:

$$(83) \frac{[A]^1}{A \rightarrow A} \mathcal{I} \rightarrow^1$$

What we've done here is taken A of (83) to represent both A and B above the line in (82), with the vertical dots representing a derivation with 0 steps, and then discharged A , producing a 'theorem' which often goes by the name 'I' (for 'Identity'). By 'theorem' we mean a formula derivable in the system that depends on no premises, since we have discharged the only premise used.

This might seem a bit puzzling, since it is perhaps not entirely clear that the rule (82) is meant to be used this way: the graphic representation of Natural Deduction falls somewhat short of full explicitness. Nevertheless, it's so convenient that people use it anyway, tending to use other formulations only when a higher degree of formal rigor is required.¹⁵

A more complicated example is this derivation of $B \rightarrow A \rightarrow C$ from the premise $A \rightarrow B \rightarrow C$:

$$(84) \frac{\frac{A \rightarrow B \rightarrow C \quad [A]^1}{B \rightarrow C} \mathcal{E} \rightarrow \quad [B]^2}{\frac{C}{A \rightarrow C} \mathcal{I} \rightarrow^1} \mathcal{E} \rightarrow \quad \frac{A \rightarrow C}{B \rightarrow A \rightarrow C} \mathcal{I} \rightarrow^2$$

If we discharge the remaining assumption, we get a theorem is called 'C', easy to remember due to its connection with Commutativity.

However one area in which the interpretation of our graphically formulated rules is not entirely clear in what is often called the 'discharge policy'. The basic question is *how many* instances of an assumption can be discharged by a single step. Consider this putative proof:

$$(85) \frac{\frac{A \rightarrow A \rightarrow B \quad [A]^1}{A \rightarrow B} \mathcal{E} \rightarrow \quad [A]^i}{\frac{B}{A \rightarrow B} \mathcal{I} \rightarrow^1} \mathcal{E} \rightarrow$$

Here we've discharged two instances of the assumption A in one step. In Classical Logic, this is allowed, but in the Linear Logic it isn't, and each $\mathcal{I} \rightarrow$ step can only discharge one instance of a given hypothesis. A consequence is the (85) is a theorem of Classical Logic but not Linear:

$$(86) (A \rightarrow A \rightarrow B) \rightarrow (A \rightarrow B)$$

Systems that have this as a theorem (called 'W') are described as 'having Contraction', on the basis that if two instances of an assumption can be used to support a conclusion, they can be contracted into one, and still support it.

¹⁵But a fully rigorous presentation of tree-style Natural Deduction is provided in Jäger (2005).

Another difference between Classical and Linear Logic is that for Classical, we need to be able to discharge hypotheses that aren't there at all:

$$(87) \frac{A}{B \rightarrow A} \mathcal{I} \rightarrow^1$$

Logics where you can do this ‘have Weakening’ (or ‘Thinning’), on the basis that a claim ‘ A ’ can be ‘weakened’ to the claim ‘if B , then A ’. To keep the symbolism and terminology confusing, the Weakening theorem ($A \rightarrow B \rightarrow A$) is called ‘K’.

Weakening causes considerable philosophical disquiet, since it is involved in various intuitively suspect inferences. For example, if we know that Mary is very busy, it seems wrong or at least misleading to say ‘If the Red Sox are winning, then Mary is very busy’, due to the likely lack of any connection between the Sox’ game and Mary’s level of busyness. There has therefore been a tradition of interest in ‘Relevant Logics’, where Weakening is unavailable. See Restall (2006) for further discussion of discharge policies.

Disallowing Weakening and Contraction means that in a proof, each premise can and must be used once, and only once. So that, for example, if your proof that the world is doomed makes two uses of the premise ‘Bush is a moron’, and does not involve Ahmedinejad at all, then you can’t conclude ‘the world is doomed’ from ‘Bush is a moron’ and ‘Ahmedinejad is a maniac’. Rather, your premises must be ‘Bush is a moron’ and ‘Bush is a moron’. This strikes many people as a rather wierd requirement to impose on proofs, but it can be made more natural by thinking of a very fussy person who (a) doesn’t want irrelevant premises in his arguments (so that they will apply as widely as possible), and (b), is interested in *how many times* each premise is used. Counting the number of times a premise is used is equivalent to using a different copy of the premise on each occasion of use, so once you’re doing that, you’re really doing linear logic.

Another popular way of thinking about linear logic is in terms of resources, for constructing things. One might regard a super-size cheeseburger, for example, as something that converts a 200-pound American into a 202-pound American, that is, something of type **big** \rightarrow **bigger**. But if we apply this to something of type **big**, then both disappear, and we’re left with just something of type **bigger**. So linear logic also also referred to as (a kind of) ‘resource-sensitive’ logic.

Philosophy aside, we certainly get a cleaner picture of tree-style deductions if we disallow both Weakening and Contraction, so that only hypotheses that appear in the tree can be discharged, and that only once. And, as a consequence, the trees become quite close in form to our glue-structures with the axiom-links contracted. One consequence of the resemblance is that the semantic reading rules we proposed for the trees apply as well to the proofs, where they are standardly called ‘labels’, and proofs that are ‘decorated’ with lambda-terms (‘proof-terms’) are called ‘labelled deductions’:

$$(88) \quad \frac{f : A \rightarrow B \quad a : A}{f(a)} \mathcal{E} \rightarrow \quad \frac{\begin{array}{c} [x : A]^i \\ \vdots \\ b : B \end{array}}{\lambda x. b : A \rightarrow B} \mathcal{I} \rightarrow$$

But there are two significant differences. Our tree structures can be regarded as reformatted deductions, but there are two kinds of possibilities for the deductions that aren't directly rendered in the tree format. One of them can be seen by considering the application of the **intersective** constructor to an adjective meaning. Consider the following instantiated versions of these (kinds of) constructors:

$$(89) \quad \lambda P. \lambda Q. \lambda x. P(x) \wedge Q(x) : (h_e \rightarrow h_p) \rightarrow (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p \\ \text{Crazy} : h_e \rightarrow h_p$$

We can construct an $\mathcal{E} \rightarrow$ proof-step as follows:

$$(90) \quad \frac{\lambda P. \lambda Q. \lambda x. P(x) \wedge Q(x) : (h_e \rightarrow h_p) \rightarrow (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p \quad \text{Crazy} : h_e \rightarrow h_p}{\lambda Q. \lambda x. \text{Crazy}(x) \wedge Q(x) : (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p} \mathcal{E} \rightarrow$$

The formulas are longish, but what's going on is pretty simple: a complex antecedent is applied to a complex premise.

But we can't do this with our tree-format, because complex arguments have to be structured as positive implications= λ -terms, which are represented as \rightarrow -introductions. If you look back at the tree-format version of (69), you'll see that the application of **intersective** to the adjective would be represented like this as an ND proof (well, actually, you'll probably have to do some figuring to really see it):

$$(91) \quad \frac{\lambda P. \lambda Q. \lambda x. P(x) \wedge Q(x) : (h_e \rightarrow h_p) \rightarrow (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p \quad \frac{\text{Crazy} : h_e \rightarrow h_p \quad [X : h_e]^1}{\text{Crazy}(X) : h_p} \mathcal{I} \rightarrow^1}{\lambda X. \text{Crazy}(X) : h_e \rightarrow h_p} \mathcal{E} \rightarrow^1 \\ \lambda Q. \lambda x. \text{Crazy}(x) \wedge Q(x) : (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p$$

As a proof, the upper-right hand part of this seems kind of dumb. We've got *Crazy* of type $g_e \rightarrow g_p$, we apply it to a hypothesis of type g_e , then discharge that hypothesis to get something of type $g_e \rightarrow h_p$ again, which, furthermore, under our understanding of the labels, does exactly the same thing as the original *Crazy*. A rather pointless 'detour', it would appear.

Whenever we find ourselves doing something like this, it would seem more sensible to eliminate the detour, and just use the proof-piece we had to begin with (here, $\text{Crazy} : h_e \rightarrow h_p$). This idea, formalized, becomes the 'proof normalization' rule of η -reduction, which says that a proof-piece that fits the format on the left should be simplified as indicated on the right:

$$(92) \quad \frac{\frac{f : A \rightarrow B \quad [x : A]^i}{f(x) : B} \mathcal{E} \rightarrow}{\frac{f(x) : B}{\lambda x. f(x) : A \rightarrow B} \mathcal{I} \rightarrow^i} \Rightarrow_{\eta} f : A \rightarrow B$$

With ND, we can use the η -reduced version of the proof, while with the glue-tree format, we can't.

If you stare at (69) for long enough, you might be able to see that the problem is that we are only allowing ourselves to run axiom-links between atomic formulas. Suppose we abandon this restrictions, and also stop insisting on expanding the meaning-constructors to glue-trees to the fullest extent possible. Then, for **intersective**, we might rest content with this partial expansion:

$$(93) \quad \begin{array}{c} g_p^- \\ \swarrow \quad \searrow \\ g_e^+ \quad g_e \rightarrow g_p^- \\ \quad \swarrow \quad \searrow \\ \quad g_e \rightarrow g_p^+ \quad (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p^- \\ \quad \quad \swarrow \quad \searrow \\ \quad \quad h_e \rightarrow h_p^+ \quad (h_e \rightarrow h_p) \rightarrow (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p^- \\ \quad \quad \quad \lambda P. \lambda Q. \lambda x. P(x) \wedge Q(x) \end{array}$$

Now, if we could run axiom links from a negative implication to a positive one, we could effect a one-step assembly like this:

$$(94) \quad \begin{array}{c} g_p^- \\ \swarrow \quad \searrow \\ g_e^+ \quad g_e \rightarrow g_p^- \\ \quad \swarrow \quad \searrow \\ \quad g_e \rightarrow g_p^+ \quad (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p^- \\ \quad \quad \swarrow \quad \searrow \\ \quad \quad h_e \rightarrow h_p^+ \quad (h_e \rightarrow h_p) \rightarrow (g_e \rightarrow g_p) \rightarrow g_e \rightarrow g_p^- \\ \quad \quad \quad \lambda P. \lambda Q. \lambda x. P(x) \wedge Q(x) \\ \quad \quad \quad \uparrow \\ \quad \quad \quad h_e \rightarrow h_p^- \\ \quad \quad \quad \text{Crazy} \end{array}$$

Well, why not? One part of the answer is that we'd have to do some formal work to make sure that we're not messing up the notion of the ϵ -correspondence that we're using, which is important for the linguistics. Another is that there's a really important sense in which it doesn't make any difference (so why spend a lot of effort trying to fix it).

It is one of the basic ideas of proof theory that there are many superficially different-looking proofs that really ought to be regarded as being the same, and the ones connected by η -reduction are good examples of this. Indeed, there are some contexts where

the ‘ η -expanded’ versions are regarded as being better, even though they are longer. In general, η -reduction is not seen as much of a bit deal.

But there’s another thing we can do with ND proofs but not our glue-trees which is considerably more significant, which is most easily approached by looking at the proof-terms. Something that you can do with lambda-calculus is first define a function with abstraction, and then apply it to something, with application. In terms of undecorated proofs, this corresponds to the following format:

$$(95) \quad \frac{\frac{\begin{array}{c} [A]^i \\ \vdots \\ B \end{array}}{A \rightarrow B} \mathcal{I} \rightarrow^i \quad A}{B} \mathcal{E} \rightarrow$$

Here, the upper left segment is the definition, the final step, the application.

In terms of proofs alone, it looks dumb to do this, because we wind up with a proof of B from A (and maybe some additional assumptions), just as we had as represented by the A and B in the upper left, connected by a vertical string of dots. That is, we derived B from A and perhaps some other assumptions, then discharged A to get $A \rightarrow B$, then applied this to A again to get B again, from A and those other assumptions.

So we should have used the ‘ β -reduced’ version of (95), which is just:

$$(96) \quad \begin{array}{c} A \\ \vdots \\ B \end{array}$$

But if we consider the proof-terms that would appear in the decorated version of β -reduction, it doesn’t look so dumb anymore.

The decorated version of (95) will be:

$$(97) \quad \frac{\frac{\begin{array}{c} [x : A]^i \\ \vdots \\ f : B \end{array}}{\lambda x. f : A \rightarrow B} \mathcal{I} \rightarrow^i \quad y : A}{(\lambda x. f)(y) : B} \mathcal{E} \rightarrow$$

So what about its β -reduced version?

Since we’re discharging the assumption A decorated with x , we want x to disappear, and the ‘obvious’ thing to do is to redo the derivation of B from A using y instead of x , so that what we wind up with for the decoration of B is f , with y substituted for x . Restating the β -reduction rule with the decorations in all their glory, we get:

$$(98) \quad \frac{\frac{\begin{array}{c} [x : A]^i \\ \vdots \\ f : B \end{array}}{\lambda x.f : A \rightarrow B} \mathcal{I} \rightarrow^i \quad y : A}{(\lambda x.f)(y) : B} \mathcal{E} \rightarrow \quad \Rightarrow_{\beta} \quad \begin{array}{c} y : A \\ \vdots \\ [y/x]f : B \end{array}$$

One important point here is that if we just grind through the steps designated by the vertical dots, using $y : A$ instead of $x : A$, $[y/x] : f$ is just what we get at the end: this fact does not depend on how we define the $[y/x]f$ notation for substitution; rather, the notation is a convenient technique for describing the fact.

Another is that although the non- β -reduced (left-hand) version is a dumb thing to do in isolation, it isn't so dumb if we have various independent uses for the $\lambda x.f$ result. That is, if there are various different things we can apply it to. For we can 'store' the possibly rather complex construction of $\lambda x.f$, and then use a single application to apply it to various different things, getting various results by following the relatively simple rules for substitution, rather than the possibly difficult creative process of coming up with f and $\lambda x.f$ from scratch each time.

Ok, well, in ND format we can produce non- β -reduced proofs, but in the glue-tree format, we can't. The insight into why is a bit harder than in the case of η -reduction. It is essentially that we don't have any way to produce a structure where a lambda-abstraction is made to apply to something. Lambda-abstractions are represented by positive implications, which appear only in 'argument position', as left-daughters to negative implications. There, if an implication appears as right-daughter of a negative implication, it is negative itself, and so is either the result of an application, or has a semantic label.

This deficiency, if it be reckoned a deficiency, can be fixed by adding another kind of link, called 'cut link', that runs from a positive implication to a negative one. But for our purposes, it's not a deficiency at all, but a virtue, because it means that the glue-tree system can only represent proofs that are in β -normal form. To see why this is good, consider how much easier elementary school arithmetic would have been if there were some convenient way to write down fractions that could only be used to produce fractions that were already in lowest terms, the way the math teacher wanted them.

Putting proofs into β -reduced form is conceptually very similar to reducing a fraction to lowest terms, since one can regard having a factor, say, 5 in the numerator and then dividing the same factor out in the denominator as a pointless detour in the production of the number designated by the fraction. But β -reduction of proofs and lambda-terms is actually a lot easier than reducing fractions, and we even have a format where non-reduced ones can't be written down at all!

More conceptually, the system is like this. Proof Theorists want to tame the bewildering

variety of different ways in which something might be proved by grouping ‘essentially equivalent’ proofs into classes, and regarding these classes as being the real proofs. We would then like to pick representatives from these classes, so that for example that if there are three classes of ‘essentially equivalent’ proofs of some proposition, we can represent the resulting three ‘essentially different’ proofs by writing down representatives, ideally chosen in some non-arbitrary fashion.

β -reduction produces a very useful classification of proofs, whereby any proofs that can be interconverted by a chain of β -reductions and reversed β -reductions are held to belong to the same class. And the β -reduced proofs are the obvious representatives (just like the lowest term form of a fraction is the obviously best representative of the rational number designated by the fraction). Now for our glue-tree scheme we also have η -reduction to think about, but here we’re not using the η -reduced forms. But this is harmless, due to the following fact. If we start with a basic predicate, say *Crazy*, and η -expand once, we get $\lambda x.Crazy(x)$. β -reduction can do nothing with this. But if we η -expand again, we get $\lambda y.(\lambda x.Crazy(x))(y)$, which is in fact subject to β -reduction, which will knock it back to $\lambda y.Crazy(y)$. But that’s just what we got after a single η -expansion (we ignore different choices for bound variables; this is technically called ‘ α -equivalence’). This means that there is a coherent notion of ‘ β -reduced, but η -expanded as much as possible’, which can be designated $\beta\eta^-$ -reduced, on the basis that something is η^- -reduced if it is η -expanded as much as possible without thereby becoming subject to β -reduction (sort of like claiming as many tax deductions as possible without incurring a high probability of getting audited).

So, the final conclusion is that the glue-tree format gives us a single representative, in $\beta\eta^-$ normal form of each possible ND proof. And these proofs, however represented, provide instructions on how to compose the meanings of the words in the utterance, to the extent that such composition can be understood as the application of functions to arguments. Which is what Montague Grammar was clearly the first kind of system to achieve for reasonable fragments of English and comparable languages, a result which glue semantics therefore has let us transfer to languages that can be given a syntactic analysis with LFG.

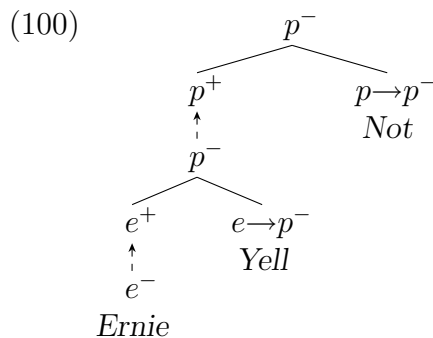
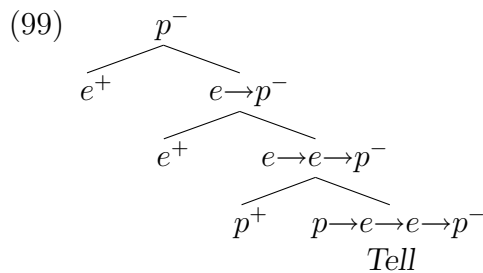
The main thing we’ve omitted from the treatment so far is ‘tensors’, which are a major ingredient in the treatment of anaphora. This, however, is a somewhat fraught topic, with a fair degree of complexity, which I leave for a different exposition that presupposes the content of this one, Andrews (2008a).

And I never said what the ‘Curry Howard Isomorphism’ was. What it is is the discovery that the system of implicational proofs, constructed with $\mathcal{E}\rightarrow$ and $\mathcal{I}\rightarrow$, has exactly the same mathematical structure as the independently invented system of typed lambda-calculus, with application and abstraction constructions, with the β and η reductions corresponding to the proof-simplification schemes which are given the same name, thanks to this discovery. These things were invented independently, in the first part of the 20th century, so that it was a real discovery that they were basically the same

thing. And the CHI holds up under various kinds of further development. For example ‘discharge policy’ for proofs is the same thing as ‘binding policy’ for lambdas: a ‘linear lambda’ can and must bind one and only one instance of its variable, while a ‘relevant lambda’ must bind at least one, but can bind as many as desired.

Such things become relevant when we start working out what is really going on in the meaning-sides of glue-constructors, an enterprise which is not very well advanced.

11 Structures for Exercises

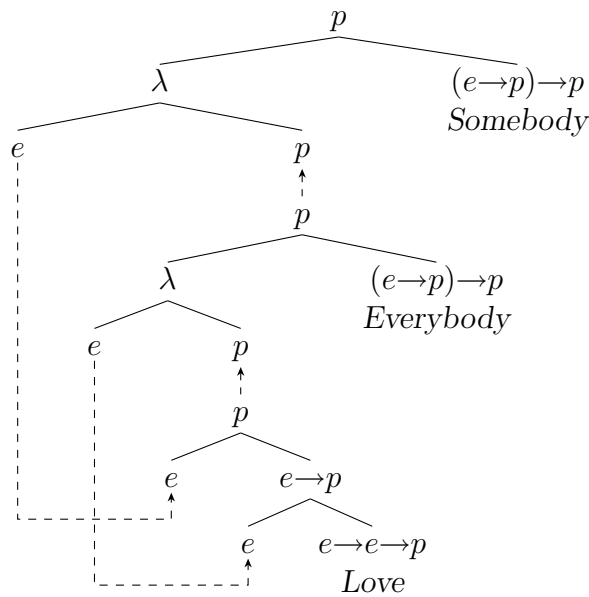


(101) a. (14) fails to satisfy (17) because the node with meaning-label *Not* is a Principle Input, but there is no dynamic path from it to the the Final Output.

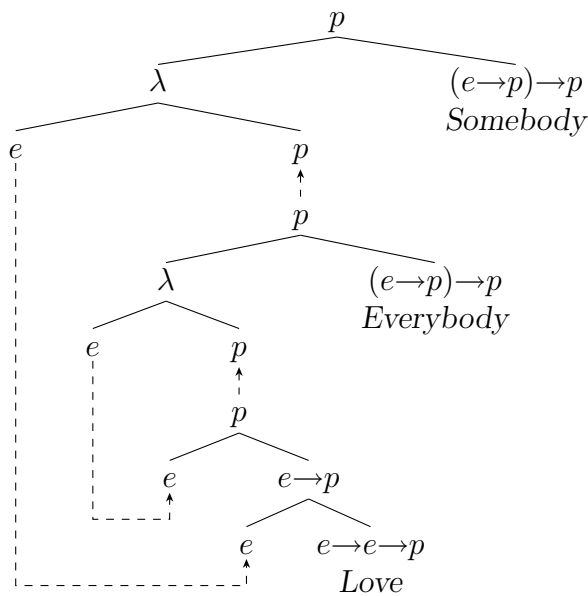
b. (15) satisfies (17), because the Principle Inputs are the nodes with the meaning-labels *Bert*, *Likes* and *Ernie*, and, from each of these, the dynamic graph provides a path to the Final Output.

(102) $\text{Not}(\text{Yell}(\text{Bert}))$.

(103) a.



b.



References

Alsina, A. 1996. *The Role of Argument Structure in Grammar*. Stanford, CA: CSLI Publications.

Alsina, A. 1997. A theory of complex predicates: Evidence from causatives in Bantu and Romance. In A. Alsina, J. Bresnan, and P. Sells (Eds.), *Complex Predicates*, 203–246. Stanford, CA: CSLI Publications.

Andrews, A. D. 2006. Semantic composition for NSM, using LFG + Glue. Proceedings of ALS2005, URL: <http://www.als.asn.au/>.

- Andrews, A. D. 2007a. Generating the input in OT-LFG. In J. Grimshaw, J. Maling, C. Manning, and A. Zaenen (Eds.), *Architectures, Rules, and Preferences: A Festschrift for Joan Bresnan*, 319–340. Stanford CA: CSLI Publications. URL: <http://arts.anu.edu.au/linguistics/People/AveryAndrews/Papers>.
- Andrews, A. D. 2007b. Glue semantics for clause-union complex predicates. In M. Butt and T. H. King (Eds.), *The Proceedings of the LFG '07 Conference*, 44–65. Stanford CA: CSLI Publications. URL: <http://csli-publications.stanford.edu/LFG/12/lfg07.html>.
- Andrews, A. D. 2008a. Propositional glue and the correspondence architecture of LFG. URL: <http://arts.anu.edu.au/linguistics/People/AveryAndrews/Papers>; uploaded to Semantics Archive.
- Andrews, A. D. 2008b. The role of PRED in LFG+glue. LFG-08 paper. URL: <http://arts.anu.edu.au/linguistics/AveryAndrews/papers>.
- Andrews, A. D., and C. D. Manning. 1993. Information-spreading and levels of representation in lfg. Technical Report CSLI-93-176, Stanford University, Stanford CA. Available at: <http://www.sultry.arts.edu.au/cmanning/papers>.
- Asudeh, A. 2004. *Resumption as Resource Management*. PhD thesis, Stanford University, Stanford CA. <http://http-server.carleton.ca/~asudeh/> (viewed June 2008).
- Asudeh, A. 2005a. Control and resource sensitivity. *Journal of Linguistics* 41:465–511.
- Asudeh, A. 2005b. Relational nouns, pronouns and resumption. *Linguistics and Philosophy* 28:375–446.
- Asudeh, A., and R. Crouch. 2002. Coordination and parallelism in glue semantics: Integrating discourse cohesion and the element constraint. In *Proceedings of the LFG02 Conference*, 19–39, Stanford, CA. CSLI Publications. URL: <http://csli-publications.stanford.edu>.
- Baez, J., and M. Stay. 2008. Physics, topology, logic and computation: A rosetta stone. URL: <http://math.ucr.edu/home/baez/rosetta.pdf>.
- Barwise, J., and R. Cooper. 1981. Generalized quantifiers and natural language. *Linguistics and Philosophy* 4:159–219.
- Casadio, C. 1988. Semantic categories and the development of categorial grammar. In E. B. R.T. Oehrle and D. Wheeler (Eds.), *Categorial Grammar and Natural Language Semantics*, 95–123. Reidel.
- Copestake, A., D. Flickinger, C. Pollard, and I. A. Sag. 2005. Minimal recursion semantics: an introduction. *Research on Language and Computation* 3:281–332.

Crouch, R., and J. van Genabith. 1999. Context change, underspecification, and the structure of glue language derivations. In Mary Dalrymple (Ed.), *Syntax and Semantics in Lexical Functional Grammar: The Resource-Logic Approach*, 117–189.

Crouch, R., and J. van Genabith. 2000. Linear logic for linguists.
URL: <http://www2.parc.com/istl/members/crouch/>.

Dalrymple, M. (Ed.). 1999. *Syntax and Semantics in Lexical Functional Grammar: The Resource-Logic Approach*. MIT Press.

Dalrymple, M. 2001. *Lexical Functional Grammar*. Academic Press.

Dalrymple, M., R. M. Kaplan, J. T. Maxwell, and A. Zaenen (Eds.). 1995. *Formal Issues in Lexical-Functional Grammar*. Stanford, CA: CSLI Publications. URL: <http://standish.stanford.edu/bin/detail?fileID=457314864>.

de Groote, P. 1999. An algebraic correctness criterion for intuitionistic multiplicative proof-nets. *TCS* 115–134. URL: <http://www.loria.fr/~degroote/bibliography.html> (viewed June 2008).

Gupta, V., and J. Lamping. 1998. Efficient linear logic meaning assembly. In *COLING/ACL 98*. Montréal. URL: <http://acl.ldc.upenn.edu/P/P98/P98-1077.pdf>.

Hindley, J. R., and J. P. Seldin. 1986. *Introduction to Combinators and λ -Calculus*. Cambridge University Press.

Jackendoff, R. S. 1973. The base rules for prepositional phrases. In S. R. Anderson and P. Kiparsky (Eds.), *A Festschrift for Morris Halle*, 345–357. Holt, Rinehard and Winston.

Jackendoff, R. S. 2002. *Foundations of Language*. Oxford University Press.

Jäger, G. 2005. *Anaphora and Type Logical Grammar*. Springer.

Kaplan, R. M. 1987. Three seductions of computational psycholinguistics. In P. White-lock, M.M.Wood, H. Somers, R. Johnson, and P. Bennet (Eds.), *Linguistics and Computer Applications*, 149–188. Academic Press. reprinted in Dalrymple et al. (1995), pp. 337–367.

Kokkonidis, M. 2008. First order glue. *Journal of Logic, Language and Information* 17:43–68. First distributed 2006; URL: <http://citeseer.ist.psu.edu/kokkonidis06firstorder.html>.

Lambek, J. 1965. The mathematics of sentence structure. *American Mathematical Monthly* 58:154–170.

Marantz, A. 1984. *On the Nature of Grammatical Relations*. Cambridge MA: MIT Press.

Melliès, P.-A. 2008. Categorical semantics of linear logic: a survey. unpublished draft, URL: <http://www.pps.jussieu.fr/~mellies/>.

Moot, R. 2002. *Proof-Nets for Linguistic Analysis*. PhD thesis, University of Utecht. URL: <http://www.labri.fr/perso/moot/> (viewed June 2008), also <http://igitur-archive.library.uu.nl/dissertations/1980438/full.pdf>.

Partee, B. H. 2006. Do we need two basic types. In H.-M. Gaertner, R. Eckardt, R. Musan, and B. Stiebels (Eds.), *Puzzles for Manfred Krifka*. Berlin. URL: www.zas.gwz-berlin.de/40-60-puzzles-for-krifka/.

Pelletier, F. J. 1999. A brief history of natural deduction. *History and Philosophy of Logic* 1–31. slightly shortened version downloadable from: [//citeseer.ist.psu.edu/pelletier98brief.html](http://citeseer.ist.psu.edu/pelletier98brief.html).

Pollard, C. 2007. Hyperintensions. To appear in *Journal of Logic and Computation*. URL: <http://www.ling.ohio-state.edu/~hana/hog/pollard2007-hyper.pdf> (viewed June 2008).

Restall, G. 2006. Proof theory and philosophy. chapters available at <http://http://consequently.org/papers/ptp.pdf>. An earlier version of this was ‘Proof and Counterexample’.

Sadler, L., and R. Nordlinger. 2008. Apposition and coordination in Australian languages: an LFG analysis. ms under review. URL: <http://privatewww.essex.ac.uk/~louisa/aal/apposition6.pdf>.