Ambiguity management in computational Glue semantics

Mark-Matthias Zymla
University of Konstanz

Proceedings of the LFG'24 Conference

Miriam Butt, Jamie Y. Findlay and Ida Toivonen (Editors)

2024

PubliKon

lfg-proceedings.org

Abstract

This paper presents extensions to XLE+Glue and the Glue semantics workbench. Concretely, it builds on Findlay & Haug's (2022) idea of multistage proving. By combining their insights with techniques first described in Lev (2007), this paper provides a more flexible implementation of multistage proving. Furthermore, it presents an extension of XLE+Glue that allows users to integrate multistage proving into computational LFG grammars. Finally, the paper discusses some insights from working on ambiguity management and semantic grammar writing suggesting that full syntactic or semantic autonomy is a challenge for computational models of LFG.

1 Introduction

The goals of this paper are two-fold.[†] Firstly, it presents a technical contribution in the shape of a new implementation of Findlay & Haug's (2022) multistage proving. This implementation tackles multiple aspects of their proposal and arguably improves on them. Secondly, it discusses the role of semantics in the projection architecture more generally. Concretely, the ideals of syntactic and semantic autonomy are put under scrutiny from the perspective of computational LFG.

Glue semantics suffers from abundant spurious and unwanted ambiguities.[1] Multistage proving is a proposal that aims to solve both of these problems to an extent (Findlay & Haug 2022). The general idea is to add additional structure to Glue proofs. Intuitively, Glue proofs are made (partially) associative (Gotham 2021). This is handled via a new projection: the proof structure, a tree-like structure partitioning Glue meaning constructors (MCs). While the idea is formally well laid out, the proposed computational implementation is somewhat rudimentary. Derivations are handled in a cascading bottom-up manner; i.e., one derivation is split up into a set of derivations. In this paper, I argue that this potentially affects Glue semantics' ability to capture long-distance dependencies. I present a computationally more adequate implementation of multistage proving that better reflects its formal ideas. Additionally, this paper presents a concrete implementation of multistage proving for the Xerox Linguistics Environment (XLE; Crouch et al. 2017), making it available for computational LFG research.

An exploration of the ideas underlying multistage proving unveils more fundamental questions about the flexibility of Glue semantics. Concretely, the question is whether Glue semantics requires us to build additional structure to properly constrain ambiguities or whether it is sufficient to transduce structure from other modules of the grammar into the semantics.[2] According to Findlay & Haug (2022), both structure building and

[†]I thank the audience of the 2024 LFG conference and the reviewers for their feedback. I am particularly thankful to Ash Asudeh who prompted this effort in 2023. Furthermore, I am grateful to Jamie Findlay for helpful discussion and comments. This work has been funded by the Deutsche Forschungsgemeinschaft (DFG) within the project CUEPAQ, Grant Number 455910360, as part of the Priority Program "Robust Argumentation Machines (RATIO)" (SPP-1999).

[1]I use the term *unwanted* ambiguities for cases where unattested readings arise to contrast them with spurious ambiguities that are semantically equivalent.

[2]This discussion is partially due to Ron Kaplan, p.c. It also builds on various other insights related to syntactic and semantic autonomy due to, among others, Asudeh (2004) and Gotham (2021).

transduction are necessary to block unwanted ambiguities, but as it stands, their proposal is more of a tool rather than a theory with any explanatory power.

Ultimately, the simple matter of fact is that a set of binary branching trees can easily represent a semantics built around function application. The goal of ambiguity management is, then, either to arrange the terminal nodes of such trees in the right order or to filter out trees in the case that multiple are possible. The question is to what extent such trees are built in the semantics, how much is projected onto the semantics by other structures, and what other factors may add additional structure to the semantics. Although it is difficult to definitively answer these questions, this paper aims to provide some new insights or at least perspectives on these issues. Thus, it covers both computational and theoretical aspects of ambiguity management in Glue semantics.

In the next section, the main challenge is introduced: spurious and unwanted ambiguities. Then, I proceed to introduce graph-based proving for Glue semantics due to Lev (2007) in Section 3. This provides the necessary background for Section 4, which introduces a graph-based approach to multistage proving. Furthermore, Section 4.4 makes a proposal for integrating the proof structure in the XLE, thus making it available for computational LFG research. In Section 5, a more broad perspective is taken exploring the syntax/semantics interface and the question of whether syntactic and semantic structures are two sides of the same coin. Finally, Section 6 concludes.

2 Compositional ambiguities in Glue semantics

Glue semantics, despite the name, is a theory of the syntax/semantics interface rather than semantics itself (Asudeh 2022).[3] This is reflected in the fact that the idea of Glue semantics is to be compatible with different syntactic and semantic representations. This is achieved by splitting up semantic representations into meaning representations, reflecting the semantics, and instructions for compositional assembly serving as an interface to the syntax. Thus, compositionality is governed by two aspects: type logic and relations to the syntax. Linear logic is a logic that allows us to capture both at the same time (Kokkonidis 2006). The relation to type logic emerges due to Montague's (1970) seminal work on formal semantics, later streamlined by Heim & Kratzer (1998). However, other than the previously cited work, Glue semantics is not based on transformations (e.g., quantifier raising) of syntactic trees.[4]

As with any other formal system for assigning linguistic representations to language, Glue semantics struggles with two types of ambiguities: unwanted ambiguities and spurious ambiguities. This can be illustrated very straightforwardly by virtue of multiple adjectival modification of nouns. Given the naive assumption that all adjectives follow the same template (that of intersective adjectives), we expect two representations for the adjectival noun phrases in (1) and (2). In the case of intersective adjectives, (1), this leads to spurious ambiguity as the two readings are equivalent. In the case of (2), we get a completely unwanted interpretation we want to rule out.

[3] Glue semantics (Dalrymple et al. 1993; Dalrymple 1999) has recently received a number of concise introductions: Asudeh (2022, 2023). Thus, we omit a re-iteration of the basics.

[4] As we will discuss later, a semantics built on function application can ultimately be represented as a binary branching tree. Glue semantics generates such trees from more syntax-independent constraints.

(1) a trustworthy Scottish chairman

a. $\lambda Q.\exists x[\text{trustworthy}(x) \wedge \text{scottish}(x) \wedge \text{chairman}(x) \wedge Q(x)] \equiv$

b. $\lambda Q.\exists x[\text{scottish}(x) \wedge \text{trustworthy}(x) \wedge \text{chairman}(x) \wedge Q(x)]$

(2) a trustworthy former chairman

a. $\lambda Q.\exists x[\text{trustworthy}(x) \wedge \text{former}(\text{chairman}(x)) \wedge Q(x)]$

b. ~~$\lambda Q.\exists x[\text{former}(\text{trustworthy}(x) \wedge \text{chairman}(x)) \wedge Q(x)]$~~

The spurious ambiguity seems innocent at first but has massive implications as sentences and analyses become more complex. That we predict unattested readings makes the situation worse. The main issue here is that it is difficult in Glue semantics to distinguish unwanted from spurious ambiguities. The reason for this is fairly simple: Ambiguities arise from the assembly instructions, i.e., linear logic, but whether an ambiguity is, in fact, wanted, unwanted, or spurious is generally determined by the meaning side. Consider the following examples. Example (3) shows a simple analysis of intersective adjectives. The intersective meaning is accounted for by conjunction on the meaning side. Since conjunction is commutative, both derivations are correct. Thus leading to spurious ambiguity. However, the meaning side and linear logic side share this property, suggesting that they are in unison.

(3) a trustworthy Scottish chairman

a. $\lambda P.\lambda x.\text{trustworthy}(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t$

b. $\lambda P.\lambda x.\text{scottish}(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t$

c. $\lambda x.\text{chairman}(x) : g_e \multimap g_t$

Derivation 1:

$$\dfrac{\text{trustworthy} : (g_e \multimap g_t) \multimap g_e \multimap g_t \qquad \dfrac{\text{scottish} : (g_e \multimap g_t) \multimap g_e \multimap g_t \quad \text{chairman} : g_e \multimap g_t}{\text{scottish(chairman)} : g_e \multimap g_t}}{\text{trustworthy(scottish(chairman))} : g_e \multimap g_t}$$

Derivation 2:

$$\dfrac{\text{scottish} : (g_e \multimap g_t) \multimap g_e \multimap g_t \qquad \dfrac{\text{trustworthy} : (g_e \multimap g_t) \multimap g_e \multimap g_t \quad \text{chairman} : g_e \multimap g_t}{\text{trustworthy(chairman)} : g_e \multimap g_t}}{\text{scottish(trustworthy(chairman))} : g_e \multimap g_t}$$

This is not always the case. We have seen in example (2) that not all adjectives work commutatively. Nonetheless, as example (4) suggests, the assembly instructions remain the same across all kinds of adjectives: they are modifiers, meaning constructors that take some input, modify it, and return it without changing its type.[5] For us, it indicates a mismatch between the meaning side and the linear logic side of meaning constructors. Glue semantics does not capture some of the finer nuances that constrain adjective ordering.[6]

[5] In actuality, some finer details change, but the main point – that adjectives are modifiers – remains the same (see Dalrymple 2001).

[6] Andrews (2018) also discusses and criticizes this treatment of adjectival modifiers. More generally, e.g, Findlay (2021) points out that the link between semantics and the rest of the projection architecture via linear logic is relatively weak, at least in the form it is currently popularly practiced.

(4) a former Scottish chairman

a. $\lambda P.\lambda x.\text{trustworthy}(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t$
b. $\lambda P.\lambda x.\text{former}(P(x)) : (g_e \multimap g_t) \multimap g_e \multimap g_t$
c. $\lambda x.\text{chairman}(x) : g_e \multimap g_t$

There are a couple of ways to go from here: Early works, e.g., Gupta & Lamping (1998) suggest leaving ambiguities of this nature underspecified, whereas Lev (2007) presents various heuristics for ambiguity management and weeding out unwanted interpretations in a computational context. More recently, Findlay & Haug (2022) suggested the introduction of additional structure to the projection architecture to deal with both unwanted and spurious ambiguities that arose from the need for calculating Glue derivations in a more computationally efficient manner.[7] Ambiguity management has also received a fair share of attention in the theoretical literature. For example, Gotham (2019, 2021) modifies the assembly language to avoid unwanted ambiguities with the goal of preserving semantic autonomy (this topic is briefly discussed in Section 5).[8]

This section has shown that the issues of spurious ambiguities and unwanted ambiguities arise from the same source. However, spurious ambiguities are mainly discussed in a computational context, whereas in theoretical work they are often swept under the rug. Thus, computational approaches, which are in pursuit of efficient systems for dealing with ambiguities, often provide a more holistic perspective. As this is the central topic of this paper, the next section introduces the more technical aspects of linear logic derivations, particularly Lev's (2007) graph-based prover, re-implemented in the Glue Semantics Workbench (Meßmer & Zymla 2018).

3 Graph-based proving

The Glue Semantics Workbench (GSWB) uses two provers based on Hepple (1996) and Lev (2007). The prover based on Hepple's work is a chart-based prover with improvements suggested by Lev (2007). The second prover is a graph-based prover based on additional work by Lev. This paper expands on the graph-based prover and focuses on the management of compositional ambiguities.

The graph-based prover can be seen as an instance of factorizing out ambiguities (Maxwell & Kaplan 1993). Intuitively, this means that derivations are partitioned into ambiguous and non-ambiguous parts such that non-ambiguous parts have to be computed only once and can participate in possibly multiple derivations resulting from ambiguity. Before delving deeper into how this is achieved, we have to discuss some core properties of graph-based proving. Firstly, we turn to the compilation process.

3.1 Compilation of premises

First proposed by Hepple (1996) for his chart prover, the compilation process has the goal of reducing all higher-order linear logic formulas into first-order formulas. A formula is higher-order if any linear implication within it has a complex antecedent. A

[7] Findlay and Haug, p.c.

[8] Ambiguity management is also sometimes mentioned more as a by-product of certain analyses, e.g., Andrews (2018); Cook & Payne (2006); Crouch & Van Genabith (1999).

common example of this is the type for quantifiers $(e \multimap t) \multimap t$. The corresponding compilation process is illustrated in (5-a). As shown there, the complex antecedent $e \multimap t$ is split up, creating a new premise $[e]^i$. This new premise corresponds to traditionally used assumptions and is marked similarly with brackets. We also highlight its origin with an index i. This index ensures that the remainder of the initial formula, here $t_i \multimap t$, combines with an element that made use of the new resource. This is shown in (5-b) where we want to combine the quantifier with a first-order formula, $e \multimap t$. First, the first-order formula consumes the compiled-out assumption $[e]^i$. The resulting t^i now carries the index of the assumption. The assumption is discharged by combining it with the assumption's original host, $t_i \multimap t$. This derivation corresponds to a simple sentence like *a dog barked*, as illustrated.

(5) a. $(e \multimap t) \multimap t \rightarrow_{comp} [e]^i, t_i \multimap t$

b.
$$\dfrac{\dfrac{\dfrac{X : [e]^i \qquad \lambda x.\text{bark}(x) : e \multimap t}{\text{bark}(X) : t^i}}{\lambda x.\text{bark}(x) : t^i} \qquad \lambda Q.\exists x[\text{dog}(x) \wedge Q(x)] : t_i \multimap t}{\exists x[\text{dog}(x) \wedge \text{bark}(x)] : t}$$

The process of compilation plays a role in accounting for long-distance dependencies in Glue derivations as there can be an arbitrary distance between the use of the assumption and its re-connection with its host.

A consequence of the compilation process is that it becomes clear that quantifiers share properties with modifiers, i.e., they can be reduced to premises of type $X \multimap X$.[9, 10] As already stated by Gupta & Lamping (1998), only modifiers cause compositional ambiguity. This means that dealing with wanted scopal ambiguities and unwanted or spurious ambiguities both rely on how modifiers are handled. The factoring-out mentioned above aims at disentangling skeletons from modifiers.[11] By factoring out modifiers from a derivation, we can constrain the need for ambiguity management to subparts of the proof. This idea was proposed by Lev (2007) for Glue semantics.

3.2 The category graph

Concretely, a so-called category graph is used.[12] A category graph for some sentence is formed by inspecting all categories that are used in its Glue derivation. Categories correspond to the set of unique linear logic formulas appearing in the premise set. They form (a part of) the vertices of the graph. During a computational derivation, the input premises are first compiled and indexed; then categories are extracted.[13] Consider example (6). From the compiled premise set, we can determine the categories in (7).

[9] The compiled out assumption ensures that quantifiers cannot arbitrarily modify any element of type t, but only those that carry the appropriate assumption. Nonetheless, their modifier status is the cause of scopal ambiguity.

[10] Impure modifiers are possible, e.g., $a \multimap b \multimap a$. They have the same properties as pure modifiers.

[11] Opposed to modifiers, skeletons are Glue premises that follow a fixed order of combination, i.e., their input must be different from their output.

[12] A more detailed explanation is given in Lev (2007). However, we state the key points here to provide a concise overview of graph-based Glue derivations.

[13] The indices are important to assure resource sensitivity and will be explained in more detail later.

Each category is unique, but it can be instantiated multiple times in MCs. For example, the category $f \multimap f$ occurs twice in the input premise set (assumption indices are not part of a category's properties). The subformula $h \multimap f$ occurs as part of the verb's meaning constructor, but is also a category in its own right. Similarly, the category f is instantiated multiple times as part of the quantifiers as well as the verb.

(6)

$\lambda Q.\forall x[\text{person}(x) \rightarrow Q(x)] : (g_e \multimap f_t) \multimap f_t$	$[0]\lambda Q.\forall x[\text{person}(x) \rightarrow Q(x)] : f_{t,[3]} \multimap f_t$
$\lambda x.\lambda y.\text{see}(x, y) : g_e \multimap h_e \multimap f_t$	$[3]X : g_e^3$
$\lambda Q.\exists y[\text{person}(y) \wedge Q(y)] : (h_e \multimap f_t) \multimap f_t$	$[1]\lambda x.\lambda y.\text{see}(x, y) : g_e \multimap h_e \multimap f_t$
	$[2]\lambda Q.\exists y[\text{person}(y) \wedge Q(y)] : f_{t,[4]} \multimap f_t$
$\longrightarrow_{\text{compile}}$	$[4]Y : h_e^4$

(7) Relevant categories:

a. g_e, h_e, f_t, $f_t \multimap f_t$, $h_e \multimap f_t$, $g_e \multimap h_e \multimap f_t$

The category graph is a directed graph built by combining categories according to the combination rules of linear logic. For example, h_e is combined with $h_e \multimap f_t$ via a combination node (the additional vertex in the graph), which points to the category f_t, indicating the obvious result of combining the two categories. This allows us to build the graph in Figure 1, on the left. There, the orange nodes correspond to the original premises before compilation.[14] Rectangular blue nodes correspond to the relevant categories used in the derivation while circular blue nodes correspond to combination steps. The yellow node corresponds to the goal category of the proof. Green nodes with multiple elements play a special role. They correspond to cycles, as with the bottommost node in Figure 1 on the left. Such cycles correspond to an embedded graph, thus, the green node on the left corresponds to the cyclic graph on the right. As indicated by the dashed lines, the inputs from the main graph on the left feed into the cyclic graph. This is due to the fact that these nodes play a role in the calculations within the embedded graph. The difference between the two graphs is to be made clear shortly.

This layout is achieved by applying a strong connectivity algorithm that condenses cycles in a graph. The result is a graph containing strongly connected components (SSCs; Tarjan 1972), sub-graphs in which there is a path between any two vertices. All modifiers are contained within strongly connected component. Thus, this allows us to treat skeleton premises and modifier premises independently. We simply need to combine skeleton premises according to the main graph and then deal with the ambiguity through the SSC with added input nodes, as shown in Figure 1 on the right.

3.3 Semantic derivation

The combination process is guided through so-called *histories*. They store information on indices and on the meaning side and keep track of combination steps via pointers to their parents. All initial categories (those whose category exactly matches an input premise) are associated with initial histories. Initial histories do not have parents, as illustrated in (8) by not having the feature p. Importantly, the category $f \multimap f$ has two histories corresponding to the two different original premises associated with it.

[14] Their outgoing edges indicate the compilation process.

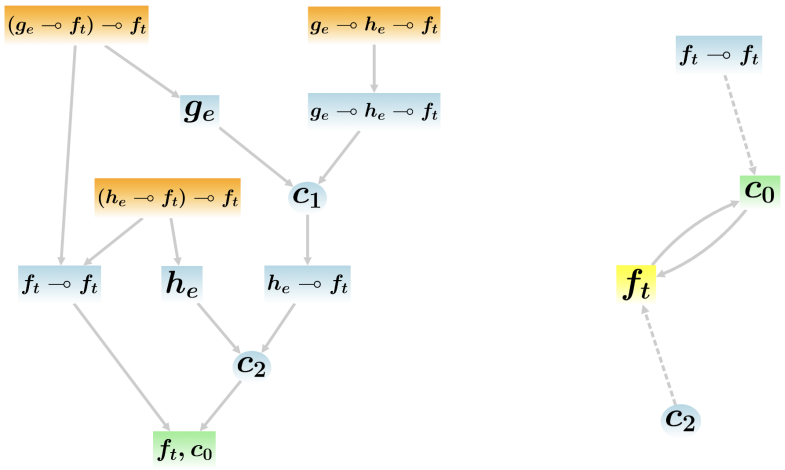

Figure 1: Linear logic derivation graph with strongly connected component in green

(8) a. History for category $g_e \multimap h_e \multimap f_t$:

$$g_e \multimap h_e \multimap f_t \quad \rightarrow \quad \{\ h_1 : [1]\ \lambda x.\lambda y.\text{see}(x, y) : g_e \multimap h_e \multimap f_t\ \}$$

b. Histories for category $f_t \multimap f_t$:

$$f_t \multimap f_t \quad \rightarrow \quad \left\{ \begin{array}{l} h_1 : [0]\ \lambda Q.\forall x[\text{person}(x) \rightarrow Q(x)] : f_{t,[3]} \multimap f_t \\ h_2 : [2]\ \lambda Q.\exists y[\text{person}(y) \wedge Q(y)] : f_{t,[4]} \multimap f_t \end{array} \right\}$$

The combination of two histories is illustrated in (9). There, the history for the category $h_e \multimap f_t$ is built up by combining the histories of the categories g_e and $g_e \multimap h_e \multimap f_t$. Accordingly, the *parents*-array marked with p provides pointers to parent histories corresponding to functor f and argument a.

For finding a successful derivation, the semantics are ignored when combining histories. Rather, they built up a tree structure consisting of function application steps. Thus, the process of finding a successful derivation is prioritized before actually building the corresponding semantics since this can be done by inspecting only the linear logic side (Lev 2007; see also Dalrymple et al. 1999a).

(9) History for category $h_e \multimap f_t$:

$$h_e \multimap f_t \quad \rightarrow \quad \left\{ \begin{array}{l} h_3 : [1,3]\ f(a) : h_e \multimap f_t, \\ p \left[\begin{array}{l} f : h_1 : [1]\ \lambda x.\lambda y.\ \text{see}(x, y) : g_e \multimap h_e \multimap f_t \\ a : h_2 : [3]\ X : g_e \end{array} \right] \end{array} \right\}$$

The full semantics of a derivation can then be calculated by tracing back function application steps from the history or histories corresponding to the goal category (yellow in Figure 1). This is illustrated in Figure 2.

Why are there two solutions? This is where the indexation of the compiled premises comes into play. Recall that the process for handling ambiguous elements is distinct from that of combining skeletons which can be simply read off the category graph. To deal with the ambiguity in example (6) we need to employ a variation of the chart prover. For this, we make the additional assumption that whenever two premises are combined,

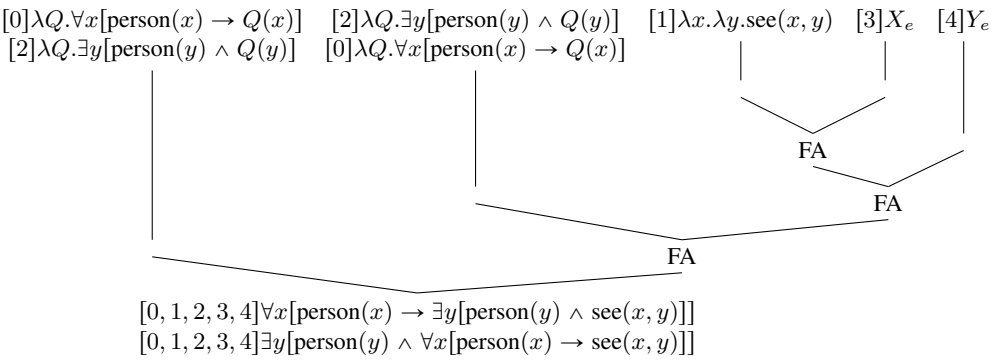

Figure 2: Resulting semantic derivation

then their index sets are combined. Furthermore, only premise sets with disjoint index sets may be combined. Example (9) illustrates this. There, the history h_3 with index set $[1, 3]$ results from the combination of the histories h_1 with index set $[1]$ and h_3 with index set $[3]$. For our running example, by combining the verb with its arguments we get the history in (10) (here with semantics for ease of exposition). These steps are the same regardless of the semantic ambiguity. We only have to compute them once!

(10) a. History for category f_t:
$f_t \quad \rightarrow \quad \{\ h_1 : [1, 3, 4]\ \text{see}(X, Y) : f_t\ \}$

This illustrates what I have stated in the previous section: the category graph disentangles skeleton and modifier premises. Example (10) is the result of the skeleton derivations of (6). For the computation of the cyclic subgraph, we use the chart prover. It works by naively trying to combine each element with each other element in the derivation until no new combinations emerge, and storing intermediate solutions on a chart. This makes it a reasonable tool for calculating proofs with multiple solutions.[15]

The factoring out of the skeleton computations results in a reduction of computations necessary for the chart prover because we can feed in results of the skeleton derivation. This is done by following the dashed lines from c_2 into the subgraph in Figure 1. This means, for the current example, we now only have to check six possible combinations (halving the number of combinations necessary with a pure chart prover).

Of these, only two succeed. The results are again naively combined with the premise set, and again, only two combinations succeed due to the disjoint index set constraint. The corresponding procedure is schematized in (11). The left column shows the input. The center column shows the intermediate results of trying to combine all initial elements on top. By attempting to combine those again with the elements below the line, we arrive at the two solutions in the right column.

[15] It is described in Hepple (1996) and Lev (2007: ch. 5), as well as Meßmer & Zymla (2018). Thus, we will not explain it in detail here. However, there is a full chart derivation of (6) in the appendix.

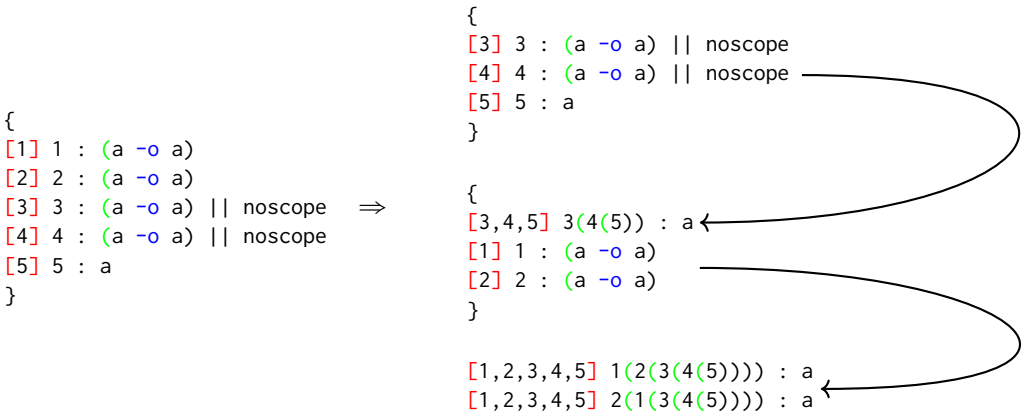

Figure 3: Partioning meaning constructors with the noscope flag

(11)
$$\begin{array}{l}[0] : (f_{t,3} \multimap f_t)\\ [2] : (f_{t,4} \multimap f_t)\\ [1,3,4] : f^{3,4}\end{array} \longrightarrow_{prove} \frac{\begin{array}{l}[1,2,3,4] : f_t\\ [0,1,3,4] : f_t\end{array}}{\begin{array}{l}[0] : (f_{t,3} \multimap f_t)\\ [2] : (f_{t,4} \multimap f_t)\\ [1,3,4] : f^{3,4}\end{array}} \longrightarrow_{prove} \begin{array}{l}[0,1,2,3,4] : f_t\\ [0,1,2,3,4] : f_t\end{array}$$

3.4 Partioning premise sets

The important caveat of the chart prover is that it is naive. In the graph-based prover, premises are sorted by the category graph, which can be built quadratically. Conversely, the chart prover is unstructured. In the worst case, its derivations are factorial. As such, even when the ambiguity is factored out, it can become overwhelming computationally.

In Section 2, we established that it is sometimes difficult in Glue to distinguish spurious ambiguities from unwanted ambiguities. However, the simplest way to deal with this is to mark MCs that would introduce spurious ambiguities as such. This simple way to deal with ambiguities is proposed by Lev (2007) in the shape of the noscope flag. The noscope flag essentially partitions the input set of the chart prover into scoping and non-scoping modifiers. Furthermore, as the name implies, noscope flags indicate that the order of application does not matter. For example, one has to find only one solution for a set containing only non-scoping modifiers (in addition to any potential arguments). The process is illustrated in Figure 3. First, the non-scoping modifiers are applied to any suitable input resources in arbitrary order.[16] The result is passed onto a second stage of chart-proving, including the result of the non-scoping derivation and the remaining scope-sensitive modifiers. Thus, two solutions are found instead of 24.

[16]This means that everything marked with noscope has more narrow scope than scoping elements. Across multiple premises marked with noscope, the scope is determined by the order in which premises are processed during the chart derivation (i.e., randomly). There are various ways of doing this efficiently but they are not important here (see, e.g., Lev 2007: 197ff.). In general, it is fairly straightforward to find a solution given the true commutativity of noscope-marked modifiers.

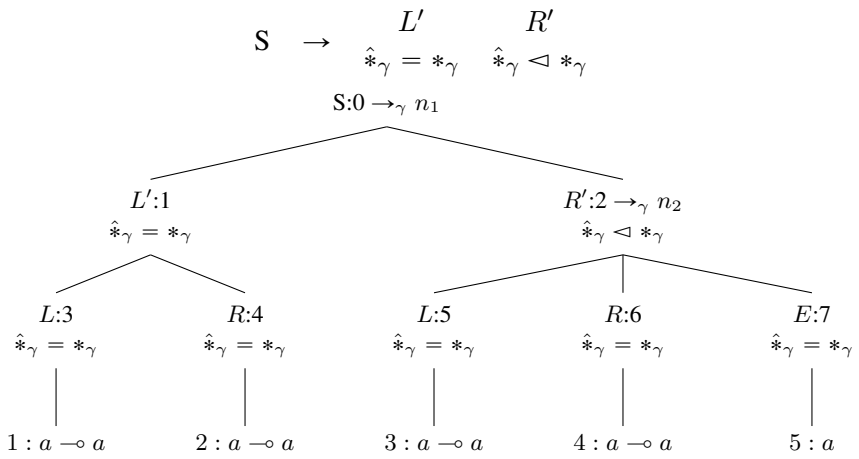

Figure 4: Example phrase structure rule and corresponding c-structure

4 A faithful implementation of multistage proving

In a sense, the idea of partitioning meaning constructors into groups has been expanded upon by Findlay & Haug (2022). While this idea is conceptually straightforward, they, furthermore, integrate it into the projection architecture of LFG in a formally sophisticated manner. In this section, we first briefly discuss their proposal, both theoretically and technically, then formulate some criticisms for their computational implementation, and finally propose a novel implementation that circumvents at least the technical problems but also may push the exploration of theoretical questions by providing an explicit implementation in the Xerox Linguistics Environment (XLE; Crouch et al. 2017).

4.1 The proof structure

Findlay & Haug (2022) propose a new projection in LFG's modular architecture, the proof structure. The proof structure is formally a tree and specified via equality and dominance constraints. This is illustrated in Figure 4. There, * identifies c-structure indices. The subscript γ maps c-structure nodes onto their proof structure counterparts. Thus, proof constraints describe relations between proof structure nodes. In Figure 4, the proof structure essentially partitions the c-structure into L' and R'. Thus, we partition our meaning constructors according to a certain c-structure configuration.[17] Let us call the resulting structure g(lue)-structure (as p(roof)-structure clashes with p(honological)-structure).

The g-structure for the example in Figure 4 is given in Figure 5.[18] There, each proof node is associated with a set of meaning constructors t_i, percolated up via the pre-terminal nodes, where i is an index co-identifying the relevant proof structure node.

[17] As a reviewer of the abstract pointed out, generally speaking, the proof structure is more flexible as it can potentially apply to indices from any projection and even introduce new nodes. We will discuss this point later.

[18] In Figure 4, the annotation indicates that only two nodes play a role in building the proof structure, namely n_1 and n_2 marked with the γ-correspondence $\rightarrow_\gamma$.

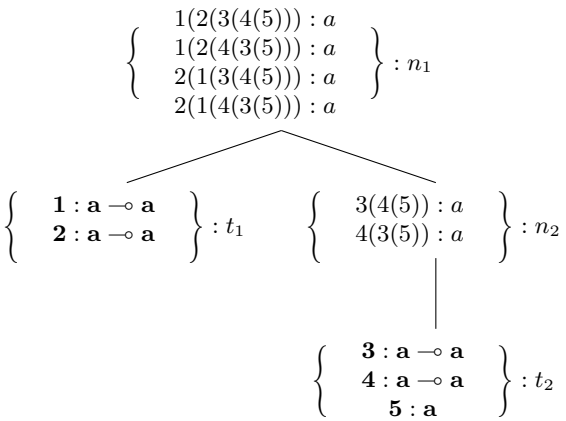

Figure 5: G-structure with added solutions

Thus, in this case, we get the given tree. By convention, the left-most daughter carries the associated meaning constructors. Consequently, our meaning constructors are partitioned into two stages. Each stage still introduces an ambiguity in our example, but once again, we have reduced 24 possible solutions to four, which could, in fact, be desired.

4.2 Original implementation and criticism

Findlay & Haug (2022) propose to implement this by virtue of a cascade of chart provers. Concretely, they traverse the tree bottom-up. For each node t_i, they calculate all possible combinations and pass up the relevant intermediate results to the next stage. Thus, in Figure 5, all the results stored in n_2 would be combined into a premise set with the elements in t_1 to calculate the solutions in n_1.

This approach has a major weakness: The intermediate goals of different stages need to be explicitly stated. This is necessary to avoid passing up information that has already been used up in the derivation. Thus, somewhere in the architecture, intermediate solutions must be specified before they are concretely derived. This seems ad hoc. That this is inelegant and not always trivial can be illustrated by examining one of their examples regarding scope freezing, e.g., example (12) (their (9) and Figure 3). There, it is assumed that it is impossible for the universal quantifier in the direct object NP to outscope the existential quantifier in the indirect object NP.

(12) Hilary gave a student every grade.
   a. $\exists < \forall$
   b. ~~$\forall < \exists$~~

This means that, similar to our previous example, we have two stages, as shown in Figure 6. The embedded node t_2 combines the verb with the quantifier scoping over the direct object. Thus, Findlay & Haug (2022) predict the intermediate and correct solution in example (13-a) for n_2 in Figure 6, as the sequent in (13-b) is valid. To work with intermediate goals computationally, they add a Goal premise with the intermediate type as antecedent. This is exemplified in (13-c) which can be then added to t_2 in

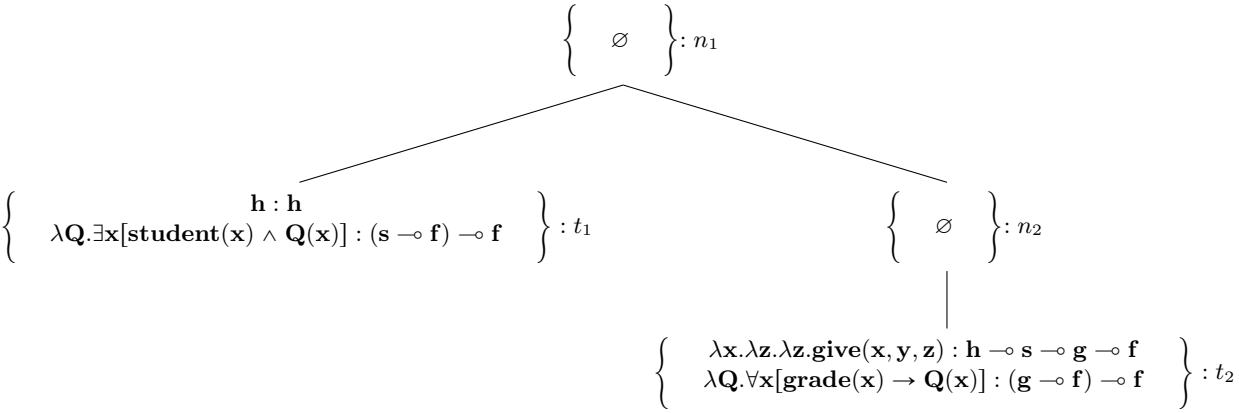

Figure 6: Computationally failing proof structure for (12)

Figure 6. The result is a set of premises that has an atomic goal G. The meaning side of G is a dummy predicate GOAL which can be stripped off the intermediate meaning representation easily.

(13) a. $\lambda x.\lambda y.\forall z[\text{grade}(z) \rightarrow \text{give}(x, y, z)]] : h \multimap s \multimap f$
b. $h \multimap s \multimap g \multimap f, (g \multimap f) \multimap f \vdash h \multimap s \multimap f$
c. $\lambda P.\text{GOAL}(P) : (h \multimap s \multimap f) \multimap G$

This approach essentially duplicates resources that are missing from t_2 by hard-coding them within the proof tree. One could argue that the information can be reconstructed from within stage n_2. However, if not hard-coded, the goal needs to be derived from the left-hand side of the sequent in (13-b), introducing additional computations and, possibly, multiple results requiring us to deal with further spurious ambiguities.[19]

Thus, overall the cascading chart prover approach to multistage proving does not faithfully implement the formal elegance of the multistage proving idea.[20] In the next section, I present an alternative method to multistage proving, which does away with the need for knowing intermediate goals beforehand, while maintaining the general idea of cascading proof steps. This is achieved by integrating multistage proving within the graph-based proving paradigm introduced in Section 3. Furthermore, I integrate this method into XLE+Glue, thus providing an explicit implementation for LFG grammars.

[19]For example, the following sequent is also valid:

(i) $h \multimap s \multimap g \multimap f, (g \multimap f) \multimap f \vdash s \multimap h \multimap f$

Thus, making the goal category of n_2 in Figure 6 ambiguous. Findlay & Haug (2022) acknowledge problems along these lines. The present proposal provides a possible solution.

[20]It is possible that this approach is expected to work in tandem with the propsal made in Findlay (2021). There, intermediate results are purposefully stored in the semantic structure. However, the computational complications discussed here still need to be considered when evaluating the implementation.

```
{
//2: Scope freezing; 1 solution
h : h_e
a-student : ((s_e -o f_t) -o f_t)
    {
    every-grade : ((g_e -o f_t) -o f_t)
    give: (h_e -o (s_e -o (g_e -o f_t)))
    }
}
```

Figure 7: Bracketed meaning constructors for example (12)

4.3 Graph-based multistage proving

In the proposal made in this paper and following Findlay & Haug's (2022) idea, multistage premise sets are represented as bracketed premise sets. Thus, Glue semantics is made partially non-associative (see also Gotham 2021). According to Moot & Retoré (2012: 111ff.), this is a desirable result: "What we would like is to have some sort of controlled access to the structural rules of associativity and commutativity."[21] An example of this is given in Figure 7.

The intuitive idea behind graph-based multistage proving is to apply the cascading chart-proving mechanism only within the cycles that may occur during graph-based proving. Let us unwrap this idea. First, we label each set of brackets according to coordinates in the proof tree and store relations between them in a separate graph structure. This is the proof structure in the original proposal, now encoded via bracketing. As explained in Section 3, graph-based proving factors out modifiers and applies chart-based proving to them. Here, we use the proof tree to partition the chart. The important difference to Findlay & Haug (2022) is that our chart is reduced. All skeleton combinations are already processed.[22] The input to the chart-prover is determined by the nodes within the strongly connected component and the nodes leading into the sub-graph representing the cycle. The graph corresponding to the MCs in Figure 7 is shown in Figure 8. Recall that the graph is based on the compiled premises shown in (14).

(14)
$$\begin{array}{l} [0] : \text{h} : h_e \\ [1] : \lambda Q.\exists x[\text{student}(x) \wedge Q(x)] : ((s_e \multimap f_t) \multimap f_t) \\ [2] : \lambda Q.\forall x[\text{grade}(x) \rightarrow Q(x)] : ((g_e \multimap f_t) \multimap f_t) \\ [3] : \lambda x.\lambda y.\lambda z.\text{give}(x,y,z) : h \multimap s \multimap g \multimap f \end{array} \longrightarrow_{\text{compile}} \begin{array}{l} [0] : \text{h} : h_e \\ [1] : (f_{t,4} \multimap f_t) \\ [2] : (f_{t,5} \multimap f_t) \\ [3] : h \multimap s \multimap g \multimap f \\ [4] : s_e \\ [5] : g_e \end{array}$$

The proof corresponding to the cycle in Figure 8 is schematized in (15). Only two input nodes are relevant as indicated by the dashed lines in the cycle representation. The algorithm collects the corresponding histories for chart-proving. Then, the premise set is partitioned according to the proof tree. The tree is traversed bottom-up from left to

[21] We will discuss commutativity briefly in Section 5.

[22] This is not always the case. Some complex cycles can contain skeleton premises which need to be taken into account. However, as the chart-proving method is a very general method for calculating Glue proofs, this is not a problem.

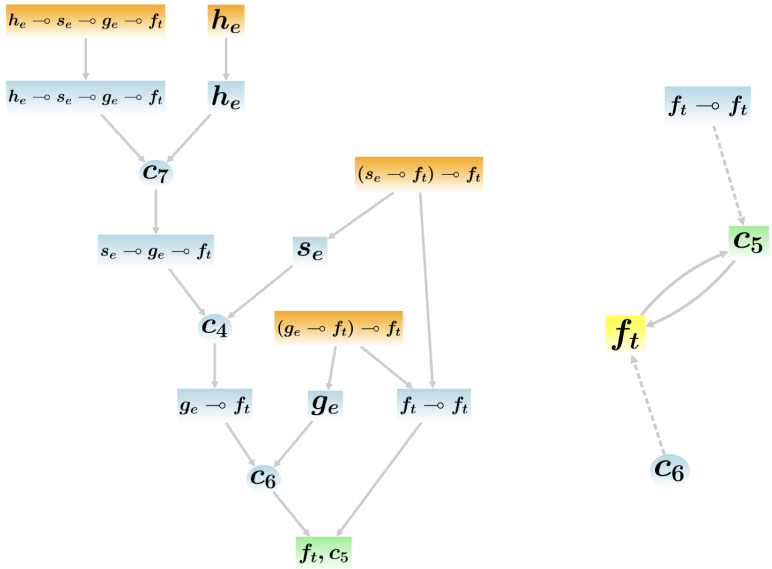

Figure 8: Derivation graph for example (14)

right.[23] As example (15) indicates, only the results of the derivation are passed on to the next stage. This is achieved by making sure that only elements are passed on where all modifiers in the input set have been applied.[24]

$$(15) \qquad \begin{array}{l}[1] : (f_{t,4} \multimap f_t) \\ [2] : (f_{t,5} \multimap f_t) \\ [0,3,4,5] : f^{4,5}\end{array} \longrightarrow_{partition} \left\{ \begin{array}{c} [2] : (f_{t,5} \multimap f_t) \\ [0,3,4,5] : f^{4,5} \\ \downarrow_{prove} \\ [1] : (f_{t,4} \multimap f_t) \\ [0,2,3,4,5] : f^{4} \end{array} \right\} \longrightarrow_{prove} \quad [0,1,2,3,4,5] : f^{4,5}$$

In summary, the method presented here does not require us to know intermediate goals due to the fact that fixed elements in the derivation are factored out and only modifiers are applied in a certain order based on the proof tree. Thus, by combining the intuitive idea of partitioning meaning constructors with a graph-based prover, we can omit certain stipulations made by Findlay & Haug (2022).

4.4 XLE+Glue with proof structure

This also makes it easier to interface multistage proving with the XLE. Concretely, we use XLE+Glue, developed by Dalrymple et al. (2020), as a basis. We extend the system with a component that extracts proof trees from XLE analyses and translates them into bracketed meaning constructor sets. We make the following basic assumptions:

[23] Sister nodes could potentially be parallelized for additional performance gains.

[24] In Section B of the appendix, a more complex example is shown that illustrates that other resources potentially need to be available at multiple stages.

```
EQUAL-GLUE(DOWN UP) =
       UP = %up
       DOWN = %down
       %down $ (t::%up ELEMENTS).

DOMINATES-GLUE(DOWN UP N) =
       @(CONCAT DAUGHTER N %daughter)
       UP = %up
       DOWN = %down
       %down $ (t::%up %daughter
                ELEMENTS).
```

$$
n_1\left[\begin{array}{ll}
\text{DAUGHTER1} & n_2\left[\begin{array}{lll} \text{ELEMENTS} & t_2\left\{\begin{array}{l} h \multimap s \multimap g \multimap f \\ (f_{t,5} \multimap f_t) \\ g_e \end{array}\right. \end{array}\right] \\
\text{ELEMENTS} & t_1\left\{\begin{array}{l} (f_{t,4} \multimap f_t) \\ s_e \\ h_e \end{array}\right.
\end{array}\right]
$$

Figure 9: Proof structure templates and sample AVM-representation for example (12)

i) The proof structure is a single rooted tree.

ii) Meaning constructors are associated with c-structure indices.

The proposal is illustrated in Figure 9. There, two templates are specified to build up the proof tree in the `t::` projection: the *equation*-template and the *dominance*-template. Both take two arguments, namely indices governed by the proof structure. The local names `%up` and `%down` are used to properly instantiate all relevant nodes with projection information, which is sometimes omitted when a node is not required for deriving the f-structure.[25] In the *dominance*-template, the additional parameter `N` is used to differentiate between sister nodes in the proof tree.[26] Consequently, the proof structure is an attribute/value matrix built from these templates. Importantly, to avoid clutter in the grammar, each node with no proof structure specification is treated as having the annotation $\hat{*}_\gamma = *_\gamma$, i.e., equality with its mother node. This is taken care of when traversing the c-structure to extract the multistage premise set.

Figure 10 illustrates a sample use of the proof structure templates to avoid unwanted ambiguities in example (14). As described in Findlay & Haug (2022), the secondary object (here OBJ2) is put in an embedded position with the verb, whereas the SUBJ and primary object remain in the dominating proof tree node. By using the same index, the embedded elements are stored as part of the same proof tree node.[27]

Bracketed premise sets are extracted by traversing XLE's c-structure output. This process relies on the fact that the c-structure is hierarchical for resolving implicit equations between mother and daughter nodes. Thus, the proof structure is also strictly a tree in this implementation. As a result, the procedure is fairly straightforward.

[25] This seems to be an idiosyncrasy of XLE rather than necessarily intended behavior. Thus, the use of local names can be seen as an implementation trick.

[26] In the current implementation, sister nodes need to be enumerated manually.

[27] Another technical trick employed here is the fact that the proof node is associated with the f-structure index of the VP rather than the c-structure index. This is due to the fact that multiple-branching trees are treated as covert binary trees in XLE. Thus, using `t::M*` to refer to the c-structure mother node instead of `t::^` would make the system a bit more complicated. However, this also illustrates how the proof structure can interact not only with c- but also with f-structure in this implementation.

```
S --> NP: (^ SUBJ)=!                    VP --> (AUX)
          (! CASE)=nom                         V: ^=!
          @(EQUAL-GLUE * t::^);                   @(DOMINATES-GLUE * t::^ 1);
      VP: (^ TNS-ASP TENSE).                      (NP: (^ OBJ)=!
                                                       (! CASE)=acc
                                                       @(EQUAL-GLUE * t::^))
                                                  (NP: (^ OBJ2) = !
                                                       @(DOMINATES-GLUE * t::^ 1))
                                                            "secondary object"
                                                    ...
```

Figure 10: Sample use of proof structure templates

4.5 Exploring multistage proving

The GSWB (Meßmer & Zymla 2018) has grown into a more comprehensive ecosystem including various aspects of computational Glue semantics that I have subsumed under the banner of XLE+Glue (coined by Dalrymple et al. 2020). The current work can be explored at https://github.com/Mmaz1988/xleplusglue/tree/lfg2024_multistage_proving. This repository combines the GSWB,[28] LiGER,[29] and the original XLE+Glue[30] into one large framework for exploring various aspects of computational Glue semantics. Furthermore, it provides a web interface for using the tools without the need for extensive technical knowledge. However, an XLE license and the XLE source code, as well as Docker,[31] are required. The repository contains a toy grammar for exploring multistage proving and a test file for the GSWB, including tests of varying complexity for the multistage proving algorithm. These tests go far beyond the simple examples discussed in this paper.

5 Some general considerations on ambiguity management

Multistage proving is, among other things, a tool for making semantic parsing more efficient. Ambiguities arise in all phases of parsing a linguistic expression. The computational complexity tends to rise as one moves from form to meaning: morphology is encoded in terms of regular languages, which can be solved in $O(n)$, c-structures are polynomial ($O(n^3)$)), and f-structures are exponential (all in the worst case). For semantics, the simple chart parser fares even worse in the worst case, being at least factorial ($O(n!)$). Consequently, it is sometimes useful to push labor toward the more computationally efficient parts of parsing. One such instance is, for example, the presence of complex categories, which effectively push functional disjuncts into the domain of phrase structure rules. Similarly, Glue semantics can be made to involve more or less information from other projections. I discuss f-structure and c-structure in particular.

[28] https://github.com/Mmaz1988/GlueSemWorkbench_v2
[29] https://github.com/Mmaz1988/liger
[30] https://github.com/Mmaz1988/xle-glueworkbench-interface
[31] https://www.docker.com/

$$\dfrac{\dfrac{\lambda x.\lambda y.\text{visit}(y,x) : g \multimap (h \multimap f) \qquad a : g}{\lambda y.\text{visit}(y,a) : h \multimap f} \qquad j : h}{\text{visit}(j,a) : f}$$

Figure 11: Traditional derivation for *Jordan visits Alex*

5.1 Imposing hierarchical structures

Linear logic is quite capable of encoding hierarchical information. This can be seen in Figure 11 based on the meaning constructors in example (16-a). They mirror the constituency hierarchy between the SUBJ and OBJ-GF in a configurational language, specifically English. Nothing in Glue semantics hinges on this possibility of mirroring the c-structure though. However, even this could still be seen as capturing the fact that there is a hierarchical relationship between SUBJ and OBJ that needs to surface somewhere in the syntax/semantics interface. If this hierarchy is already expressed at the level of syntax, why not transduce it to the semantics from there?

(16) Jordan visits Alex.

a. **John** $j : g$
Alex $a : h$
visits $\lambda x.\lambda y.\text{visit}(y,x) : g \multimap (h \multimap f)$

Arguably, this hierarchy is not always enforced in syntax. This has been famously shown in Austin & Bresnan (1996) for Australian Aboriginal languages and has also been explored computationally, for example, in Urdu (Butt & King 2007). The languages in question are modeled with an exocentric category S that dominates a flat c-structure (see also Kroeger 1993). In this case, the semantics would not simply recapitulate hierarchical information from the syntax but rather integrate the hierarchical information directly into the semantics. This is at odds with the idea of a framework that is often sold on the basis of getting away without structure building (in contrast to, e.g., the logical form (LF) approach to semantics; Heim & Kratzer 1998). However, ultimately, it seems to be a question of implicit vs. explicit hierarchical ordering. In other words, hierarchical structures always emerge. We can either make them explicit, or they will occur implicitly during the derivation process.

To understand this, let us compare the hierarchical approach above to a flatter approach: since the rising popularity of event semantics (Davidson 1967; Parsons 1990), Glue semantics has developed in a direction that eliminates meaningful hierarchical structure from Glue proofs by treating (some) arguments as modifiers. This development perhaps began around the time of Asudeh & Giorgolo (2012), where only optional arguments are treated as modifiers. It also follows a more general trend in formal semantics away from the traditional saturation-based semantics (i.e., the approach outlined above) towards a restriction-based semantics (Chung & Ladusaw 2003).

Just recently, radically modifier-oriented approaches to Glue semantics have found their way into computational LFG. Two recent works highlight this.[32] One of them is on

[32]Lev (2007) also already uses event semantics, but there core arguments of verbs are still hard-coded

$$\dfrac{\lambda P.\lambda x.P \wedge \text{ag}(e) = x : f \multimap g \multimap f \qquad \dfrac{\dfrac{\lambda P.\lambda x.P \wedge \text{th}(e) = x : f \multimap h \multimap f \qquad \text{visit}(e) : f}{\lambda x.\text{visit}(e) \wedge \text{th}(e) = x : h \multimap f} \qquad a : h}{\text{visit}(e) \wedge \text{th}(e) = a : f}}{\dfrac{\lambda x.\text{visit}(e) \wedge \text{th}(e) = a \wedge \text{ag}(e) = x : g \multimap f}{\text{visit}(e) \wedge \text{th}(e) = a \wedge \text{ag}(e) = j : f} \qquad j : g}$$

$$\dfrac{\lambda P.\lambda x.P \wedge \text{th}(e) = x : f \multimap h \multimap f \qquad \dfrac{\dfrac{\lambda P.\lambda x.P \wedge \text{ag}(e) = x : f \multimap g \multimap f \qquad \text{visit}(e) : f}{\lambda x.\text{visit}(e) \wedge \text{ag}(e) = x : g \multimap f} \qquad j : g}{\text{visit}(e) \wedge \text{ag}(e) = j : f}}{\dfrac{\lambda x.\text{visit}(e) \wedge \text{ag}(e) = j \wedge \text{th}(e) = x : h \multimap f}{\text{visit}(e) \wedge \text{ag}(e) = j \wedge \text{th}(e) = a : f} \qquad a : h}$$

Figure 12: Simplified event semantics derivations for *Jordan visits Alex*

coordination (Przepiórkowski & Patejuk 2023), implementing a Champollion (2015)-style semantics. The other is actually Findlay & Haug (2022), discussed in this paper, which implements a version of Asudeh et al.'s (2014) restriction-based event semantics.

Example (17) presents a simplified set of meaning constructors for such a restriction-based semantics.[33] As shown there, the meaning of the verb is assembled from its core meaning, i.e., the eventuality it describes, and its arguments, the entities participating in the eventuality which are treated as modifiers (one of the fundamental properties of Parsons's 1990 neo-Davidsonian event semantics). As a result, each argument introduces additional spurious ambiguities (see Figure 12).

(17)

John	$j : g$
Alex	$a : h$
visits	$\text{visit}(e) : f$
	$\lambda P.\lambda x.P \wedge \text{ag}(e) = x : f \multimap g \multimap f$
	$\lambda P.\lambda x.P \wedge \text{th}(e) = x : f \multimap h \multimap f$

Computationally, we can use something like the `noscope` flag mentioned in Section 3.4 to avoid spurious ambiguity. However, what this does is simply force an arbitrary hierarchy for applying modifiers. After all, we privilege one hierarchical ordering over the other when we choose a proof tree from the two possibilities in Figure 12.[34] In Glue semantics, linear logic is simply a vehicle for constraining the lambda calculus and, thus, function application forests.

Consequently, a hierarchy will emerge in either case. The only question is whether enforcing explicitly the hierarchy makes predictions that are not borne out. In many cases, the main difference seems to be descriptive parsimony (cf. Asudeh et al. 2014).

in the meaning constructors of the verb.

[33] We leave e as a free variable to keep the types concise. Of course, in actuality, they would be lambda abstracted over and bound by an existential closure mechanism.

[34] Computationally speaking, linear logic has the benefit that it is not actually necessary to enumerate the two possible solutions (i.e., this means they would be constructed and then filtered), but rather, only one solution can be calculated without ever expecting another one. Critically, this only works in commutative contexts, i.e., non-scoping contexts.

However, the relationship between descriptive parsimony and computational efficiency is not always clear. What is clear, though, is that computational implementations profit from explicit structure building to weed out spurious ambiguities.

As the above discussion suggests, the important question could be where the structure is built. Tools like parameterized rules suggest that disambiguating early can increase performance. Similarly, Zymla (2024) argues that, from a computational perspective, it is easier to leave completeness and coherence in f-structure, as unnecessary computations are avoided in the semantics by filtering out possible but unwanted analyses early. Overall, the idea should be to build structure early where possible. This is exactly what multistage proving allows us to do. Thus, the fact that it has arisen as part of the development towards a modifier-based event semantics is not all that surprising. An alternative view is that multistage proving is another instance of factoring out calculations (as is the graph prover for the chart prover). This view highlights an alternative strategy for efficient ambiguity management: distribute complex facts across multiple simple projections. This, of course, is in the spirit of LFG (Kaplan 1995).

5.2 Extensions of linear logic

The proposal made in this paper presupposes the *implicational fragment* of linear logic. There are at least two typical extensions that have been explored in the literature: the implicational fragment with quantification over variables of type t and *multiplicative linear logic*. The first was and is famously used to model the flexibility of quantifiers, and the second has been used in various guises to generate compound objects. Do these extensions complicate the current view on ambiguity management?

(18) $\forall X_t.(e_e \multimap X_t) \multimap X_t \quad \Leftrightarrow \quad (e_e \multimap \%scope_t) \multimap \%scope_t$,
where often $\%scope$ = $(GF+ \uparrow)$

(19) a. $\lambda x.\lambda y.\text{visit}(x, y) : g \multimap h \multimap f \quad \Leftrightarrow \quad \lambda\langle x \times y\rangle.\text{visit}(x, y) : g \otimes h \multimap f$

b. $\lambda z.z \times z : (\uparrow \text{ ANTECEDENT}) \multimap ((\uparrow \text{ ANTECEDENT}) \otimes \uparrow)$

The first extension is the de facto standard in Glue semantics theory and is illustrated in (18). There, two versions of the familiar quantifier type $(e \multimap t) \multimap t$ are shown. The quantifier on the left freely scopes over any constants of type t. In comparison, the quantifier on the right only scopes over constants that lie on an inside-out functional uncertainty (IOFU) path (i.e., only constants that dominate the quantifier). Ultimately, the two approaches are likely equivalent in terms of resulting analyses for quantifier types. Eliminating quantification over linear logic variables from the used Glue fragment is, in fact, an instance of resolving ambiguities early that is particularly popular in computational Glue (for reasons of efficiency, as discussed in the previous section). However, it is sometimes dispreferred theoretically as it obfuscates semantic autonomy (Gotham 2021; but see, e.g., Andrews 2010 for arguments in favor of the IOFU-approach).

The second extension is the inclusion of multiplicative conjunction. From a computational perspective, the addition of multiplicative conjunction $\otimes$ can be reduced to linear implication in cases like (19-a) (Hepple 1998). However, it also has the interesting aspect of allowing us to duplicate meanings, e.g., to copy antecedents of pronouns

(Dalrymple et al. 1999b).[35] Lev (2007: ch. 8.2) shows that such cases can be covered by the implicational fragment of linear logic, given careful consideration. Furthermore, Lev (2007) argues that such approaches are difficult to maintain as it is not possible to rule out spurious ambiguities in corresponding Glue derivations. He concludes that pronoun resolution should be left to a pragmatic module rather than solved during semantic composition (see also Kokkonidis 2006; Dalrymple et al. 2018). Given the view defended in Zymla (2024), I am inclined to take a similar stance.

Overall, the extensions discussed here introduce additional complications from a computational perspective without much payoff. It also seems like they can be ultimately reduced to clever uses of simple function application. Thus, we can maintain the position that function application is what lies at the heart of formal semantics.

6 Conclusion

The main goal of this paper is to provide a more nuanced approach to ambiguity management in computational Glue semantics under examination of the recent multistage proving proposal by Findlay & Haug (2022). To this end, the paper presents a more faithful implementation that simplifies the computational machinery by eliminating some adhoc requirements, particularly pre-specified goals. This is achieved by integrating the original idea into Lev's (2007) graph-based prover. This also allows us to use some other tools of the graph-based prover, e.g., the `noscope` flag that essentially eliminates commutatively equivalent analyses, considerably increasing efficiency. Thus, we can relax or constrain Glue semantics derivations across two dimensions of combinatory logic (Moot & Retoré 2012). However, the exact configuration based on meaningful semantic generalizations needs to be baked into the formal tools outlined by Findlay & Haug (2022) and explored in the present paper.

Concretely, as Gotham (2021) explains by virtue of quantifiers, different semantic properties, e.g., semantic monotonicity, may affect scope interactions (see also Lev 2007: 194ff.). Thus, the simple mechanisms of equality and dominance used for building proof structures may need to be made sensitive to an intricate set of constraints in the vein of Gotham (2019, 2021). Next to these concerns, phenomena like the exceptional scopal properties of indefinites (Farkas 1981; Brasoveanu & Farkas 2011) may also require more intricate mechanisms for constraining scope. Thus, there is room for further research regarding the proof structure and multistage proving.

This paper aims to steer research in this direction by providing a computational implementation of proof structure and multistage proving in XLE+Glue. Through this, hopefully, computational Glue and theoretic innovation will stay in touch, harboring LFG's strength of a close relationship between theoretical and computational research.

References

Andrews, Avery D. 2010. Propositional Glue and the correspondence architecture of LFG. *Linguistics and Philosophy* 33(3). 141–170.

[35] The example given in (19-b) is due to Asudeh (2005).

Andrews, Avery D. 2018. Sets, heads, and spreading in LFG. *Journal of Language Modelling* 6(1). 1–53.

Asudeh, Ash. 2004. *Resumption as resource management*. Ph.D. thesis, Stanford University.

Asudeh, Ash. 2005. Relational nouns, pronouns, and resumption. *Linguistics and Philosophy* 28. 375–446.

Asudeh, Ash. 2022. Glue semantics. *Annual Review of Linguistics* 8. 321–341. https://doi.org/10.1146/annurev-linguistics-032521-053835.

Asudeh, Ash. 2023. Glue semantics. In *Handbook of Lexical Functional Grammar*, 651–697. Berlin: Language Science Press. https://doi.org/10.5281/zenodo.10185963.

Asudeh, Ash & Gianluca Giorgolo. 2012. Flexible composition for optional and derived arguments. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG'12 Conference*, 64–84. Stanford, CA: CSLI Publications. https://typo.uni-konstanz.de/lfg-proceedings/LFGprocCSLI/LFG2012/papers/lfg12asudehgiorgolo.pdf.

Asudeh, Ash, Gianluca Giorgolo & Ida Toivonen. 2014. Meaning and valency. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG'14 Conference*, 68–88. Stanford, CA: CSLI Publications. http://web.stanford.edu/group/cslipublications/cslipublications/LFG/19/papers/lfg14asudehetal.pdf.

Austin, Peter & Joan Bresnan. 1996. Non-configurationality in Australian Aboriginal languages. *Natural Language & Linguistic Theory* 14(2). 215–268.

Brasoveanu, Adrian & Donka Farkas. 2011. How indefinites choose their scope. *Linguistics and Philosophy* 34. 1–55.

Butt, Miriam & Tracy Holloway King. 2007. Urdu in a parallel grammar development environment. *Language Resources and Evaluation* 41. 191–207.

Champollion, Lucas. 2015. The interaction of compositional semantics and event semantics. *Linguistics and Philosophy* 38(1). 31–66.

Chung, Sandra & William A. Ladusaw. 2003. *Restriction and saturation* (Linguistic Inquiry Monographs 42). Cambridge, MA: The MIT Press.

Cook, Philippa & John Payne. 2006. Information structure and scope in German. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG'06 Conference*, 141–161. Stanford, CA: CSLI Publications. https://web.stanford.edu/group/cslipublications/cslipublications/LFG/11/pdfs/lfg06cookpayne.pdf.

Crouch, Dick, Mary Dalrymple, Ronald M. Kaplan, Tracy Holloway King, John T. Maxwell III & Paula Newman. 2017. *XLE documentation*. Palo Alto Research Center. https://ling.sprachwiss.uni-konstanz.de/pages/xle/doc/xle_toc.html.

Crouch, Richard & Josef Van Genabith. 1999. Context change, underspecification, and the structure of glue language derivations. In Mary Dalrymple (ed.), *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*, 117–189.

Dalrymple, Mary. 1999. *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*. Cambridge, MA: The MIT Press.

Dalrymple, Mary. 2001. *Lexical Functional Grammar* (Syntax and Semantics 34). San Diego, CA: Academic Press.

Dalrymple, Mary, Vaneet Gupta, John Lamping & Vijay Saraswat. 1999a. Relating resource-based semantics to categorial semantics. In Mary Dalrymple (ed.), *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*, 261–280.

Dalrymple, Mary, Dag T. T. Haug & John J. Lowe. 2018. Integrating LFG's binding theory with PCDRT. *Journal of Language Modelling* 6(1). 87–129. https://doi.org/10.15398/jlm.v6i1.204.

Dalrymple, Mary, John Lamping, Fernando Pereira & Vijay Saraswat. 1999b. Quantification, anaphora, and intensionality. In Mary Dalrymple (ed.), *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*, 39–89.

Dalrymple, Mary, John Lamping & Vijay Saraswat. 1993. LFG semantics via constraints. In *Proceedings of the Sixth Conference on European Chapter of the Association for Computational Linguistics (EACL '93)*, 97–105. USA: Association for Computational Linguistics. https://doi.org/10.3115/976744.976757.

Dalrymple, Mary, Agnieszka Patejuk & Mark-Matthias Zymla. 2020. XLE+Glue – a new tool for integrating semantic analysis in XLE. In Miriam Butt & Ida Toivonen (eds.), *Proceedings of the LFG'20 Conference*, 89–108. Stanford, CA: CSLI Publications. https://web.stanford.edu/group/cslipublications/cslipublications/LFG/LFG-2020/lfg2020-dpz.pdf.

Davidson, Donald. 1967. The logical form of action sentences. In Nicholas Rescher (ed.), *The logic of decision and action*, 81–120. Pittsburgh, PA: University of Pittsburgh Press.

Farkas, Donka F. 1981. Quantifier scope and syntactic islands. In *Papers from the Seventeenth Regional Meeting of the Chicago Linguistic Society*, 59–66. Chicago, IL: Chicago Linguistic Society.

Findlay, Jamie Y. 2021. Meaning in LFG. In I. Wayan Arka, Ash Asudeh & Tracy Holloway King (eds.), *Modular design of grammar: linguistics on the edge*, 340–374. Oxford, UK: Oxford University Press.

Findlay, Jamie Y. & Dag T. T. Haug. 2022. Managing scope ambiguities in Glue via multistage proving. In Miriam Butt, Jamie Y Findlay & Ida Toivonen (eds.), *Proceedings of the LFG'22 Conference*, 144–163. Konstanz, Germany: PubliKon. https://lfg-proceedings.org/lfg/index.php/main/article/view/18.

Gotham, Matthew. 2019. Constraining scope ambiguity in LFG+Glue. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG'19 Conference*, 111–129. Stanford, CA: CSLI Publications. https://web.stanford.edu/group/cslipublications/cslipublications/LFG/LFG-2019/lfg2019-gotham.pdf.

Gotham, Matthew. 2021. Approaches to scope islands in LFG+Glue. In Miriam Butt, Jamie Y. Findlay & Ida Toivonen (eds.), *Proceedings of the LFG'21 Conference*, 146–166. Stanford, CA: CSLI Publications. https://web.stanford.edu/group/cslipublications/cslipublications/LFG/LFG-2021/lfg2021-gotham.pdf.

Gupta, Vineet & John Lamping. 1998. Efficient linear logic meaning assembly. In *Proceedings of the 17th International Conference on Computational Linguistics*, vol. 1, 464–470. Association for Computational Linguistics. https://doi.org/10.3115/980845.980924.

Heim, Irene & Angelika Kratzer. 1998. *Semantics in generative grammar*. Oxford, UK: Blackwell.

Hepple, Mark. 1996. A compilation-chart method for linear categorial deduction. In *Proceedings of the 16th Conference on Computational Linguistics*, vol. 1, 537–542. Association for Computational Linguistics. https://aclanthology.org/C96-1091.

Hepple, Mark. 1998. Memoisation for Glue language deduction and categorial parsing. In *Proceedings of the 17th International Conference on Computational Linguistics*, vol. 1, 538–544. Association for Computational Linguistics. https://doi.org/10.3115/980845.980935.

Kaplan, Ronald M. 1995. Three seductions of computational psycholinguistics. In *Formal issues in Lexical-Functional Grammar*, 339–367. Stanford, CA: CSLI Publications.

Kokkonidis, Miltiadis. 2006. A Glue/λ-DRT treatment of resumptive pronouns. In Janneke Huitink & Sophia (eds.), *Proceedings of the Eleventh ESSLLI Student Session*, 51–63.

Kroeger, Richard. 1993. *Phrase structure and grammatical relations in Tagalog*. Ph.D. thesis, Stanford University.

Lev, Iddo. 2007. *Packed computation of exact meaning representations*. Ph.D. thesis, Stanford University.

Maxwell, John T, III & Ronald M. Kaplan. 1993. The interface between phrasal and functional constraints. *Computational Linguistics* 19(4). 571–590.

Meßmer, Moritz & Mark-Matthias Zymla. 2018. The Glue semantics workbench: a modular toolkit for exploring linear logic and Glue semantics. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG'18 Conference*, 249–263. Stanford, CA: CSLI Publications. https://web.stanford.edu/group/cslipublications/cslipublications/LFG/LFG-2018/lfg2018-messmer-zymla.pdf.

Montague, Richard. 1970. English as a formal language. In Bruno Visentini (ed.), *Linguaggi nella società e nella tecnica*, 189–224. Edizioni di Communita.

Moot, Richard & Christian Retoré. 2012. *The logic of categorial grammars: a deductive account of natural language syntax and semantics* (Lecture Notes in Computer Science 6850). Heidelberg: Springer.

Parsons, Terence. 1990. *Events in the semantics of English*. Cambridge, MA: The MIT Press.

Przepiórkowski, Adam & Agnieszka Patejuk. 2023. Filling gaps with Glue. In Miriam Butt, Jamie Y. Findlay & Ida Toivonen (eds.), *Proceedings of the LFG'23 Conference*, 223–240. Konstanz, Germany: PubliKon. https://lfg-proceedings.org/lfg/index.php/main/article/view/41.

Tarjan, Robert. 1972. Depth-first search and linear graph algorithms. *SIAM Journal on Computing* 1(2). 146–160.

Zymla, Mark-Matthias. 2024. Computational challenges from theoretical semantic modeling in LFG. Unpublished manuscript.

A Proof procedures

Complex chart-based prover example

$\lambda Q.\forall x[person(x) \rightarrow Q(x)] : (g_e \multimap X_t) \multimap X_t$
$\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$
$\lambda Q.\exists y[person(y) \wedge Q(y)] : (h_e \multimap Y_t) \multimap Y_t$

$\longrightarrow_{compile}$

Agenda	Chart
$[0]\lambda Q.\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$	
$[3]X : g_e^3$	
$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$	
$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$	
$[4]Y : h_e^4$	

...

Agenda	Chart
$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$	$[3]X : g_e^3$
$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$	$[0]\lambda Q.\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$
$[4]Y : h_e^4$	

Agenda	Chart
$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$	$[3]X : g_e^3$
$[4]Y : h_e^4$	$[0]\lambda Q.\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$
$[1,3]\lambda y.see(X,y) : (h_e \multimap f_t)^3$	$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$

...

Agenda	Chart
$[1,3]\lambda y.see(X,y) : (h_e \multimap f_t)^3$	$[3]X : g_e^3$
	$[0]\lambda Q.\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$
	$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$
	$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$
	$[4]Y : h_e^4$

Agenda	Chart
$[1,3,4]see(X,Y) : f_t^{3,4}$	$[3]X : g_e^3$
	$[0]\lambda Q.\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$
	$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$
	$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$
	$[4]Y : h_e^4$
	$[1,3]\lambda y.see(X,y) : (h_e \multimap f_t)^3$

Agenda	Chart
$[0,1,3,4]\forall x[person(x) \rightarrow see(x,Y)] : f_t^4$	$[3]X : g_e^3$
$[1,2,3,4]\exists y[person(y) \wedge see(X,y)] : f_t^3$	$[0]\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$
	$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$
	$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$
	$[4]Y : h_e^4$
	$[1,3]\lambda y.see(X,y) : (h_e \multimap f_t)^3$
	$[1,3,4]see(X,Y) : f_t^{3,4}$

Agenda	Chart
$[0,1,2,3,4]\exists y[person(y) \wedge \forall x[person(x) \rightarrow see(x,y)]] : f_t$	$[3]X : g_e^3$
$[0,1,2,3,4]\forall x[person(x) \rightarrow \exists y[person(y) \wedge see(x,y)]] : f_t$	$[0]\forall x[person(x) \rightarrow Q(x)] : X_{t,[3]} \multimap X_t$
	$[1]\lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t$
	$[2]\lambda Q.\exists y[person(y) \wedge Q(y)] : Y_{t,[4]} \multimap Y_t$
	$[4]Y : h_e^4$
	$[1,3]\lambda y.see(X,y) : (h_e \multimap f_t)^3$
	$[1,3,4]see(X,Y) : f_t^{3,4}$
	$[0,1,3,4]\forall x[person(x) \rightarrow see(x,Y)] : f_t^4$
	$[2,1,3,4]\exists y[person(y) \wedge see(X,y)] : f_t^3$

...

□

B Non-atomic modifier example – multiple adjectives

(20) A big black dog appeared.

```
{
//10: adjective test; 1 solution
appear : (7_e -o 11_t)
a : ((9_e -o 8_t) -o ((7_e -o 11_t) -o 11_t))
    {
    big : ((9_e -o 8_t) -o (9_e -o 8_t))
        {
        black : ((9_e -o 8_t) -o (9_e -o 8_t))
        dog : (9_e -o 8_t)
        }
    }
}
```

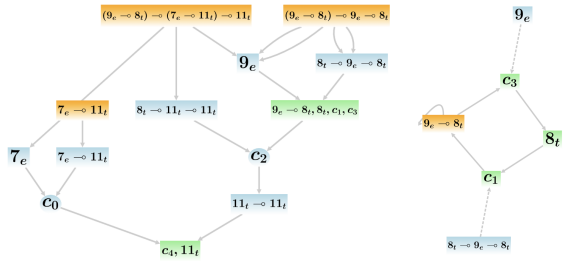

Figure 13: Meaning constructors and derivation graph for example (20)

(21)

$$
\begin{array}{l}
[0] : \lambda x.dog(x) : 9_e \multimap 8_t \\
[1] : \lambda P.\lambda Q.\exists x[P(x) \wedge Q(x)] : (9_e \multimap 8_t) \multimap (7_e \multimap 11_t) \multimap 11_t \\
[2] : \lambda P.\lambda x.[black(x) \wedge P(x)] : (9_e \multimap 8_t) \multimap 9_e \multimap 8_t \\
[3] : \lambda P.\lambda x.[big(x) \wedge P(x)] : (9_e \multimap 8_t) \multimap 9_e \multimap 8_t \\
[4] : \lambda x.appear(x) : 7_e \multimap 11_t
\end{array}
\quad \longrightarrow_{compile} \quad
\begin{array}{l}
[0] : 9_e \multimap 8_t \\
[1] : 8_t \multimap 11_t \multimap 11_t \\
[2] : 8_t \multimap 9_e \multimap 8_t \\
[3] : 8_t \multimap 9_e \multimap 8_t \\
[4] : 7_e \multimap 11_t \\
[5] : 9_e \\
[6] : 7_e \\
[7] : 9_e \\
[8] : 9_e
\end{array}
$$

(22)

$$
\begin{array}{l}
[2] : 8_{t,7} \multimap 9_e \multimap 8_t \\
[3] : 8_{t,8} \multimap 9_e \multimap 8_t \\
[0,5] : 8^5 \\
[0,7] : 8^7 \\
[0,8] : 8^8 \\
[5] : 9_e \\
[7] : 9_e \\
[8] : 9_e
\end{array}
\quad \longrightarrow_{partition} \left\{
\begin{array}{l}
\downarrow_{prove} \\
[3] : 8_{t,8} \multimap 9_e \multimap 8_t
\end{array}
\right\} \longrightarrow_{prove}
$$

$$
\begin{array}{l}
[2] : 8_{t,7} \multimap 9_e \multimap 8_t \\
[0,5] : 8^5 \\
[0,7] : 8^7 \\
[0,8] : 8^8 \\
[5] : 9_e \\
[7] : 9_e \\
[8] : 9_e
\end{array}
$$

$$
\begin{array}{l}
[0,2,5,7] : 8_t \\
[0,2,7,8] : 8_t \\
[5] : 9_e \\
[7] : 9_e \\
[8] : 9_e \\
[0,5] : 8^5 \\
[0,7] : 8^7 \\
[0,8] : 8^8
\end{array}
$$

$[0,2,3,5,7,8] : 8$

•Due to the disjoint index-set constraint and the need to discharge assumptions, only the blue elements may be combined into a single solution.