Formal Semantics for Dependency Grammar

Dag T. T. Haug and Jamie Y. Findlay
Department of Linguistics and Nordic Studies
University of Oslo

Abstract

In this paper, we provide an explicit interface to formal semantics for Dependency Grammar, based on Glue Semantics. Glue Semantics has mostly been developed in the context of Lexical Functional Grammar, which shares two crucial assumptions with Dependency Grammar: lexical integrity and allowance of non-binary-branching syntactic structure. We show how Glue can be adapted to the Dependency Grammar setting and provide sample semantic analyses of quantifier scope, control infinitives and relative clauses.

1 Introduction

Although the name *Dependency Grammar* suggests a theory covering everything that could reasonably be understood as *grammar* (often, these days, phonology, morphology, syntax and semantics), it is fair to say that the focus has to a large extent been on *syntax*. Nevertheless, there have been some attempts to extend the idea to phonology (e.g. Dresher and van der Hulst 1998) and semantics (e.g. the tectogrammatical layer of Functional Generative Description: Sgall et al. 1986). In Section 2, we argue that such frameworks lack some important desiderata of semantic theories and suggest that it is reasonable for Dependency Grammar to remain agnostic about semantics and instead attempt to build an interface between dependency syntax and established semantic theories. The main contribution of the paper is to provide such an interface to one influential semantic theory, compositional (also known as formal or logical) semantics in the tradition going back to Frege. We take inspiration from the implementation described in Gotham and Haug (2018), but the focus here is on how Glue Semantics can provide a general interface to semantics for Dependency Grammar, irrespective of this concrete implementation that is tied to a particular meaning language (partial CDRT, Haug 2014) and a particular version of Dependency Grammar (Universal Dependencies, de Marneffe et al. 2021), which deviates from most theoretical versions of Dependency Grammar in various respects.

In Section 3 we briefly introduce compositional semantics and the constraints it puts on the interface to syntax. Then we introduce Glue Semantics as a way of satisfying those constraints in Section 4. Finally, in Section 5 we show how Glue Semantics can be applied to dependency syntax. Section 6 concludes.

2 Previous work

The fundamental concept of Dependency Grammar is of course *dependencies*. But these are in themselves nothing but asymmetric, binary relations as we find them in many domains. For example, a phrase structure tree can be defined in terms of two such relations, dominance and precedence. The characteristic feature of dependency syntax is therefore not just that it is based on dependencies, but that those dependencies are taken to hold between *words*.[1] This can be seen as a strong version of the Lexical Integrity Hypothesis (Bresnan and Mchombo, 1995): not only are words atomic with respect to syntax, but they are the only atoms of syntax.

One intuitive way to extend dependency syntax to semantics, therefore, is to find an analogue to words on which to build semantic graphs. Indeed, Koller et al. (2019) provide a useful classification of graph-based semantic representations by the degree to which the nodes of the graph are anchored in the words of the sentence: some representations, such as CCG word-word dependencies (Hockenmaier and Steedman, 2007), just use the words as nodes; others, such as Prague Tectogrammatical Graphs (Zeman and Hajic, 2020) allow for a looser correspondence where nodes can also rep-

[1]Obviously we can also define notions derived from word-word dependencies, such as the transitive closure of the dominance relation, yielding something similar to constituents.

resent elided material (e.g. pro-drop, ellipsis), or copied material (e.g. words that are interpreted twice in a coordination structure); and yet others, most prominently Abstract Meaning Representations (Banarescu et al., 2013) are fully unanchored: there is no explicit correspondence between words and nodes.

While such frameworks have shown themselves useful for various computational tasks, including natural language inference, we argue that they currently lack two features that a semantic theory should have: compositionality and an explicit proof theory.

By compositionality we mean that, given a representation of the syntax (in our case, a dependency graph) and of the lexical items in the sentence, it should be possible to enumerate the possible semantic representations of the sentence. This is a weak notion of compositionality: we do not require that there are interpretations of parts of the dependency graph, nor that syntax and lexicon determine a unique meaning, only that there is some form of systematicity in the mapping between complete syntactic and semantic representations. Notice that this is a theoretical desideratum rather than a practical one: given a large dataset of hand-annotated semantic representations, it might make more sense to train a semantic parser directly rather than going via syntax, and this is in fact a common approach in natural language inference these days. Nevertheless it is clear that if we want to construct a semantic *theory* for Dependency Grammar, the semantic representations must be constrained by the syntactic representations we assume if the theory is to have any empirical bite. And yet not all graph-based semantic theories have this: Abstract Meaning Representation, for example, is hand-annotated without regard to any particular syntactic representation. While this does not make it less useful, it does make it hard to use as a semantic theory for Dependency Grammar. The Prague tectogrammatical layer, on the other hand, is a graph-based semantic representation that is explicitly linked to a surface dependency syntax representation, the analytical layer. Similarly, Meaning-Text Theory develops an interface between syntax and a graph-based semantic representation (see Kahane 2003 for an introduction).

By explicit proof theory, we mean that the semantic representations must be able to answer questions like "if a set of sentences P is true, does it follow that sentence h is true?". We take the ability to answer such questions to be a core property of human reasoning. Again, this is a theoretical desideratum: in natural language inference tasks, we are typically only given a few explicit sentences p_1, p_2 from P, whereas an inference to h relies on implicit propositions $p_3, \ldots, p_n$, which could be either just world knowledge or somehow be made salient/likely by the explicit premises p_1, p_2. In this situation, rather than trying to enumerate the possible background knowledge on which an inference may draw, it may easier to predict directly whether p_1 and p_2 make h likely. But this cannot be the basis for a semantic theory.

We are not aware of any graph-based semantic frameworks that provide a sound and complete inference system for computing entailments, though some come close. Graphical Knowledge Representation (Kalouli and Crouch, 2018; Crouch and Kalouli, 2018) explicitly views "graphs as first-class semantic objects that should be directly manipulated in reasoning and other forms of semantic processing". The semantic graphs of Meaning-Text theory are more directed towards tasks like paraphrasing rather than logical deduction, but Kahane (2005) explores the connection to logic.

Indeed, basing semantic representations on logic is one straightforward way to provide a proof theory. This is a long tradition reflected in many theories such as Montague's intensional logic (Montague, 1973), Discourse Representation Theory (Kamp and Reyle, 1993) and Minimal Recursion Semantics (Copestake et al., 2005). Linking dependency syntax to this line of work therefore provides the advantage of being able to connect to a large body of semantic work. But to do this, we must solve the compositionality problem: how do we systematically build formulae in some logic-based formalism from a dependency graph? To our knowledge, Dependency Tree Semantics (Robaldo, 2006) was the first attempt to provide such an interface between dependency syntax and formal semantics. However, Robaldo only deals with quantifiers and quantifier scope ambiguity, and it is not obvious how to generalize his work to other phenomena. The aim of this paper, then, is to provide a general solution to the compositionality problem which would allow dependency syntacticians to connect their syntactic analyses to existing work in formal semantics, or indeed to develop their own semantic analyses in parallel with syntax.

3 Formal semantics and the syntax-semantics interface

As we pointed out above, basing semantic theory on logic provides an immediate proof theory. Indeed, the very development of formal logic from Aristotle onwards can be seen as a way to provide a proof theory for natural language. The real problem, then, is compositionality: how do we systematically constrain the logical formulae that are licit translations of a given natural language sentence? One influential way to achieve this is to provide meanings for lexical items and let the syntactic structure of the sentence guide how we assemble them into a meaning for the whole. This is known as Frege's principle of compositionality (although it is not clear that Frege endorsed it in this form): the meaning of a (syntactically complex) whole is a function only of the meanings of its (syntactic) parts together with the manner in which these parts were combined. This is a much stronger notion of compositionality than what we saw in Section 2, but it has guided much previous work in formal semantics.

One immediate problem is that it is not always clear what the meanings of the parts should be. For example, it seems intuitive that the meaning of *Every man loves Chris* is something like $\forall x.man(x) \to love(x, c)$. Here it seems obvious that the verb *loves* contributes the predicate $love$, the word *man* contributes the predicate man, and the name *Chris* provides the constant c; but that then leaves the determiner *every* to contribute the rest of the meaning, i.e. the quantifier $\forall x$ and the ocurrences of x that it binds, as well as the implication $\to$, although these parts are scattered around in the sentence in a way which makes it unclear how we can provide a systematic procedure for combining the meanings.

Yet Montague's (1973) insight was that the lambda calculus *can* provide such a systematic procedure. Intuitively, the scattered meaning of *every* can be represented as $\forall x.? \to ?$, where the two question marks represent predicates containing x. In the lambda calculus we can represent this as $\lambda P.\lambda Q.\forall x.P(x) \to Q(x)$. This means that *every* is a function that takes two predicates and says that for any x, if the first predicate (the noun P that *every* combines with) applies, then the second predicate (the verb that *every P* is an argument of) also applies.

Montague's system based on the lambda calculus achieves compositionality, but it imposes strong constraints on the syntax-semantics interface that are problematic from the point of view of Dependency Grammar.

First, the *homomorphism problem*: compositionality in the strict sense requires that syntax and lexicon jointly *determine* meaning: meaning differences between two sentences must be attributed either to the parts of the sentences (i.e. the lexicon), or the manner in which they are combined (i.e., the syntax). Therefore, if a sentence with no ambiguous words is semantically ambiguous, that difference must necessarily be reflected in the syntax. This is the case, for example, with different quantifier scopings. More generally, homomorphism requires that the syntactic tree is strictly binary branching, which is typically not the case in dependency structures. For example, the lambda calculus requires that a verb combine with its subject and object in a particular order (it must combine with one before the other), whereas Dependency Grammars typically assume no hierarchical difference between subject and object, with both being sister nodes under the verb.

One way to go would be to use the *syntactic function* to distinguish the two, for example by replicating the view of most phrase structure grammars that the object bears a closer relation to the verb than the subject. This is the approach taken in UDepLambda (Reddy et al., 2017), where a syntactic function hierarchy is used to binarize the dependency tree before it is fed to the composition process. But given that many languages exhibit subject-object scope ambiguities, it makes more sense to interpret the flat dependency tree as an underspecified representation, which entails giving up on the view that syntax and lexicon determine meaning.

Second, *lexical integrity* is another problem. It is often natural that single lexical items provide two or more different meanings that do not directly combine with each other, but interact with other elements of the sentence in complex ways. For example, the verb introduces the basic predicate-argument structure, but in many languages also temporal and modal meanings, and we cannot necessarily just combine these first: modal meanings, for example, may need to take scope over the arguments of the verb. If these composition patterns are to be directly determined by the syntax, we need to assume abstract syntactic heads for modality, tense etc. This is indeed often done in Chomskyan approaches, but is alien to dependency syntax, which normally assumes lexical integrity, i.e. that words are the atoms of the syntactic structure.

In sum, "standard" formal semantics in the tradition after Montague relies on strictly binary syntax and syntactic decomposition of lexical items. This makes it hard to adapt to Dependency Grammar. But fortunately these problems have been tackled within the tradition of another lexicalist theory of syntax that also does not enforce binary syntax, namely Lexical Functional Grammar (Kaplan and Bresnan, 1982; Dalrymple et al., 2019), via the theory of the syntax-semantics interface called Glue Semantics (Glue: Dalrymple et al., 1993; Asudeh, 2022).

4 Basic Glue Semantics

The semantic building blocks in Glue are called meaning constructors; these are expressions consisting of two parts: a meaning, given in the lambda calculus over some formal language; and a formula of another logic, so-called linear logic (Girard, 1987), which constrains but does not necessarily uniquely determine the valid patterns of combination between meaning constructors. Semantic composition is logical deduction, driven by the linear logic parts of meaning constructors.

Before we get into the technical details of how this works, let us consider how it helps with the problems just described. Treating semantic composition as logical deduction helps to loosen the conection between meaning composition and syntax: provided the logic we use has the property of commutativity, then the order in which we combine meanings is driven wholly by the types of the meanings themselves, and not by the order the words they correspond to happen to occur in the string. Since we therefore no longer require the syntax itself to impose a strict order of combination, it also frees us from the obligation to limit our syntactic trees to binary branching ones. Finally, it means that semantic ambiguities, such as scope ambiguities, need not correspond to syntactic ambiguities: since the order of combination in syntax and semantics can vary independently, there can be semantic ambiguities which have no syntactic correlate. We will present an explicit example of how this works shortly.

Linear logic is chosen as the logic of semantic combination because it has the property of resource sensitivity: premises cannot be reused or discarded in linear logic, unlike in classical logic. This is because linear logic lacks the structural rules of Weakening and Contraction (Restall, 2000). If a logic contains the rule of Weakening, then premises can be freely added; this is shown schematically below, where A and B represent individual premises, and Γ represents a set of premises:

$$\frac{\Gamma \vdash B}{\Gamma, A \vdash B}$$

That is, if we can prove B from the set of premises in Γ, we can also prove it from Γ and some other premise A. If a logic contains the rule of Contraction, extra occurences of a premise can be freely discarded:

$$\frac{\Gamma, A, A \vdash B}{\Gamma, A \vdash B}$$

That is, if we can prove B from Γ and two instances of A, we can also prove it from Γ and just one instance of A.

By removing these rules, a logic becomes resource sensitive in the sense that premises are resources that must be kept track of and accounted for: they can be "used up" in a way that is not the case in classical logic. This is evident in the behaviour of implication, for example. If we apply the rule of *modus ponens* in classical logic, we can prove not only the consequent of the conditional, but also retain both of the premises in the conclusion if we so wish:

$$\begin{array}{rll} A, A \rightarrow B & \vdash & B \\ & \vdash & A \wedge (A \rightarrow B) \wedge B \end{array}$$

By contrast, in linear logic, *modus ponens* uses up both premises in proving the consequent, so that the consequent alone is left over ($\multimap$ is linear implication, and $\otimes$ is multiplicative conjunction, which for present purposes can be thought of as the linear logic equivalent of $\wedge$):

$$\begin{array}{rll} A, A \multimap B & \vdash & B \\ & \nvdash & A \otimes (A \multimap B) \otimes B \end{array}$$

All of this means that, as Dalrymple et al. (1999, 15) put it, premises in linear logic "are not context-independent assertions that may be used or not", as in classical logic, but rather "*occurrences* of information which are generated and used exactly once" (emphasis in original). This seems to be a good

Application : implication elimination

$$\frac{f : A \multimap B \qquad a : A}{f(a) : B} \multimap_{\mathcal{E}}$$

Abstraction : implication introduction

$$\frac{\begin{matrix}[\mathbf{x_1} : A]^1 \\ \vdots \\ f : B\end{matrix}}{\lambda x.f : A \multimap B} \multimap_{\mathcal{I},1}$$

Figure 1: Correspondences between operations in the lambda calculus and proof rules in linear logic

fit with how linguistic meaning contributions behave, since they too are resource sensitive (Asudeh, 2012, ch. 5). For example, the sentence *Naomi loves James* cannot have the meaning $love(n, n)$ (i.e. 'Naomi loves herself'), where we ignore the meaning of *James* and use the meaning of *Naomi* twice.

Returning to Glue Semantics, the linear logic formulae of the meaning constructors contributed by the words of a sentence (and sometimes also by structural properties of the sentence) are premises which must all be used up in constructing a proof of the linear logic formula corresponding to the sentence as a whole. Thanks to the correspondence between rules of logical deduction and operations in the lambda calculus known as the Curry-Howard isomorphism (Curry and Feys, 1958; Howard, 1980), each step in this proof also provides instructions for what to do with the meaning expression that forms the other part of a meaning constructor, thus providing us with the compositional semantics we are looking for.

The two most important rules of linear logic which we make use of are those of implication elimination (i.e. *modus ponens*) and implication introduction (i.e. hypothetical reasoning). In the lambda calculus, these correspond to the operations of function application and lambda abstraction, respectively, as shown in Figure 1. In these proofs, meaning constructors are written with a colon separating the meaning expression on the left-hand side and the linear logic formula on the right-hand side. As mentioned above, the symbol $\multimap$ represents linear implication. Figure 2 gives a more linguistic example, combining a transitive verb with its two arguments. We use the following conventions when writing proofs: unannotated

$$\frac{\dfrac{\lambda x.\lambda y.love(x, y) : A \multimap B \multimap C \qquad n : A}{\lambda y.love(n, y) : B \multimap C} \qquad j : B}{love(n, j) : C}$$

Figure 2: Glue proof for *Naomi loves James*

proof steps correspond to implication elimination, and β-reduction is performed silently. For now, we continue to use arbitrary labels for the atoms in the linear logic formulae; in Section 5 we will see how these can be connected (or 'glued') to the syntax.

As mentioned, one of the strengths of the logical deduction approach to semantic composition is that scope ambiguities need not correspond to syntactic ambiguities. Instead, they emerge from the fact that distinct proofs can (sometimes) be obtained from the same set of premises. Figure 3 shows an example of this phenomenon for the scopally ambiguous sentence *Someone loves everyone*. From the same three lexical premises (shown in labelled boxes), we can obtain two distinct proofs, via hypothetical reasoning, where the quantifiers scope in different orders: on top, we see the proof of the surface scope reading ('there is a person who loves everyone'), where *everyone* is applied before *someone*, so that the latter scopes over the former; below, we see the inverse scope reading ('everyone is loved by someone'), where the quantifiers are applied in the opposite order. Both possibilities are afforded by the linear logic, without requiring different premises to begin with, which means that the same syntactic analysis can serve for both readings.

5 Application to Dependency Grammar

The next question that arises is: how do we connect the linear logic formulae in meaning constructors to such a syntactic analysis, specifically in a Dependency Grammar setting? In the previous section, we simply used atomic formulae like A and B, but we cannot assume that the lexical entries of e.g. a verb and its subject know each other's types absolutely. Instead, the formulae must be made relative, and based on the syntactic structure.

There are several ways this can be done: here we adopt so-called "first-order Glue" (Kokkonidis, 2007). As the name says, this is a first-order logic, where the predicates are type-constructors and the terms are nodes of the syntactic trees. By convention, we use type constructors that are mnemonic for the corresponding Montagovian type on the lambda calculus side, so that e.g. E takes a tree

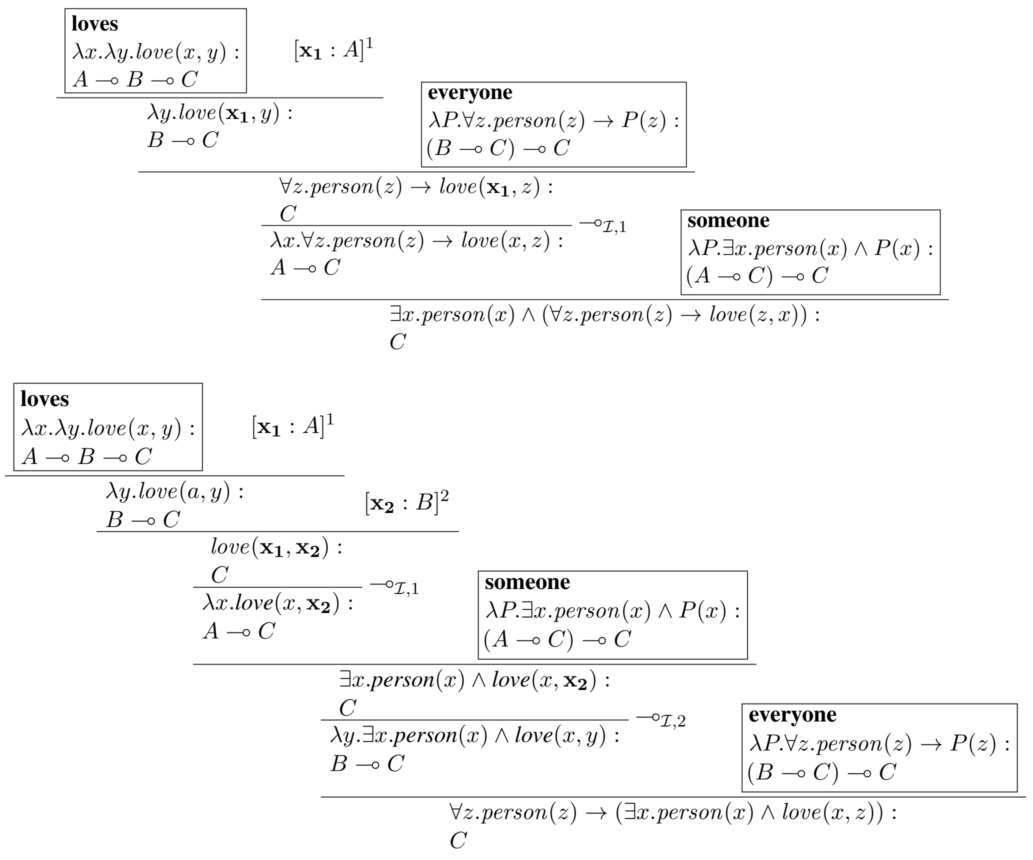
loves
$\lambda x.\lambda y.love(x,y) : A \multimap B \multimap C$
$[\mathbf{x_1} : A]^1$
$\lambda y.love(\mathbf{x_1}, y) : B \multimap C$
everyone
$\lambda P.\forall z.person(z) \to P(z) : (B \multimap C) \multimap C$
$\forall z.person(z) \to love(\mathbf{x_1}, z) : C$
$\multimap_{\mathcal{I},1}$
$\lambda x.\forall z.person(z) \to love(x,z) : A \multimap C$
someone
$\lambda P.\exists x.person(x) \wedge P(x) : (A \multimap C) \multimap C$
$\exists x.person(x) \wedge (\forall z.person(z) \to love(z,x)) : C$
loves
$\lambda x.\lambda y.love(x,y) : A \multimap B \multimap C$
$[\mathbf{x_1} : A]^1$
$\lambda y.love(a,y) : B \multimap C$
$[\mathbf{x_2} : B]^2$
$love(\mathbf{x_1}, \mathbf{x_2}) : C$
$\multimap_{\mathcal{I},1}$
$\lambda x.love(x, \mathbf{x_2}) : A \multimap C$
someone
$\lambda P.\exists x.person(x) \wedge P(x) : (A \multimap C) \multimap C$
$\exists x.person(x) \wedge love(x, \mathbf{x_2}) : C$
$\multimap_{\mathcal{I},2}$
$\lambda y.\exists x.person(x) \wedge love(x,y) : B \multimap C$
everyone
$\lambda P.\forall z.person(z) \to P(z) : (B \multimap C) \multimap C$
$\forall z.person(z) \to (\exists x.person(x) \wedge love(x,z)) : C$

Figure 3: Glue proofs for the two readings of *Someone loves everyone*

node and constructs a linear logic type that corresponds to something of type e on the meaning side. The type of a noun, then, can be given as $E(\hat{*}) \multimap T(\hat{*})$, where $\hat{*}$ refers to the node that the noun occupies.

In order to make this work, it is not enough to be able to refer to the word's own node with $\hat{*}$; we must also be able to refer to the syntactic context to make sure that entries "fit together". Here we exploit the fact that sets of paths through the dependency tree can be expressed through regular expressions over the alphabet $\mathcal{L} \cup \{\uparrow\}$, where $\mathcal{L}$ is the set of syntactic labels and $\uparrow$ refers to the mother node. For convenience we also use $\hat{*}$ as a start symbol. For example, assuming standard labels, $\hat{*}$ SUBJ in a lexical entry refers to that node's subject daughter, $\hat{*}$ OBJ to the object daughter, $\hat{*}$ (SUBJ|OBJ) to the set of the subject and object daughter, and $\hat{*}$ COMP* SUBJ to the set of SUBJ daughters embedded under zero or more COMP daughters.

For the case of the transitive sentence involving quantifiers that we saw in Section 4, we can use the full lexical entries in Figure 4. The terms of the linear logic (i.e. the arguments to the type constructors E and T) are paths through the tree. Given a syntactic tree with numbered nodes, they can be *instantiated* to numbers. Assume that *someone*, *loves* and *everyone* are numbered 1, 2, 3 and that 1 and 3 are SUBJ and OBJ daughters of 2 respectively. Then the type of *someone* becomes $(E(1) \multimap T(2)) \multimap T(2)$, *loves* gets the type $E(1) \multimap E(2) \multimap T(2)$ and *everyone* $(E(3) \multimap T(2)) \multimap T(2)$. It is easily seen that these types are isomorphic to the atomic types we used in Figure 3 and so the same proofs go through.

This technique can be extended to deal with elements that are semantically active but not present in the syntactic structure. First, consider the case of pro-drop. Many dependency grammarians take the view that pro-dropped subjects should not be represented in the syntax, but they are obviously semantically active. To deal with this, we allow paths in lexical entries to be *constructive*, i.e. if a path does not lead to a node in the tree, we construct the path, making sure we pick unused numbers for the implicit nodes we need. If a second path references the same node that did not exist in the syntax, we use this number to reference it. To avoid infi-

someone	$\lambda P.\exists x.person(x) \wedge P(x)$	:	$(E(\hat{*}) \multimap T(\uparrow)) \multimap T(\uparrow)$
loves	$\lambda x.\lambda y.love(x, y)$	:	$E(\hat{*}\ \text{SUBJ}) \multimap E(\hat{*}\ \text{OBJ}) \multimap T(\hat{*})$
everyone	$\lambda P.\forall x.person(x) \to P(x)$	:	$(E(\hat{*}) \multimap T(\uparrow)) \multimap T(\uparrow)$

Figure 4: Lexical entries for *Someone loves everyone*

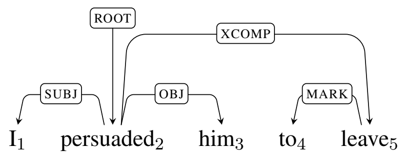

Figure 5: Control infinitive

nite trees, path parts under the Kleene star are not interpreted constructively.

When we allow constructive paths, we can deal with pro-dropped subjects by assuming that all verbs that require a subject also introduce a meaning constructor of type $E(\hat{*}\ \text{SUBJ})$ or its Montague lift $(E(\hat{*}\ \text{SUBJ}) \multimap T(\hat{*}\ \uparrow)) \multimap T(\hat{*}\ \uparrow)$.[2] The meaning side will depend on the semantics for anaphora that is adopted (if the pro-drop subject is anaphoric, as is usually the case).

Let us now look at the slightly more complicated example of control infinitives. These too have an implicit subject position which must be represented in the syntax, but which is often not made explicit in a dependency syntax tree. We will see how Glue Semantics makes it possible to nevertheless give a semantic analysis.

A sample syntactic structure is given in Figure 5, with lexical entries and the linear logic proof in Figures 6–7. The fact that the object of *persuade* is also interpreted as the subject of the infinitive is encoded in the meaning of *persuade* by the fact that the variable y occurs in both positions. Crucially, the constructive interpretation of paths allows us to identify $E(\hat{*}\ \text{SUBJ})$ in the meaning constructor of *leave* and $(E(\hat{*}\ \text{XCOMP SUBJ})$ in the constructor of *persuade*, even if *leave* has no subject in the syntactic representation.

Finally, let us look at the more complicated example of relative clauses. We first consider English relative clauses of the type *the dog that they thought we admired*. Figure 8 show two ways of analyzing such sentences that are found in the Dependency Grammar literature, differing in how *that* is attached. The two dashed lines show the main options: either it is attached as an object of *admire*, or it is attached to *think* as a subordinator.

The first type of annotation makes explicit where the gap inside the relative clause is. The second does not; and if *that* is left out, there is in any case no way of indicating where the gap is in a surface-oriented dependency analysis. In settings such as Gotham and Haug (2018), where no lexical information is assumed, it is crucial to know where the gap is, but in the present, theoretical Dependendcy Grammar setting, we can assume we have access to valency information telling us that *admire* takes an object and *think* does not; therefore there is no ambiguity in where the gap is.[3]

From a semantic point of view, the ordinary analysis of relative clauses in formal semantics, which we follow here, is that they denote properties or, extensionally speaking, sets. That is *dog* denotes the set of dogs, *that they thought we admired* denotes the set of things that they thought we admired, and *dog that they thought we admired* denotes the intersection of these sets, i.e. the entities that are dogs and that they thought we admired. The precise semantic analysis is not our agenda here, but a reasonable interpretation of *that they thought we admired* (simplifying away from tense and intensionality) would be $\lambda x.think(a_1, admire(s_+, x))$ where a_1 is a free variable representing the anaphor *they* and s_+ is a constant referring to a group including the speaker.

The lexical entries are given in Figure 9. Notice that *that* is semantically vacuous (whether analysed as a subordinator or an object). When these meanings are instantiated with the node numbers from Figure 8, we can construct the proof in Figure 10. To save space we do not show how *admire* and *think* combine with their subjects. Notice how node 9 is introduced, since the dependency tree has no object daughter of *admire*. This does not imply any commitment to an empty category in the syntax: the only role of this element is to provide abstraction over the gap in the relative clause. This

[2]This meaning constructor can be optional, since it will not be possible to construct a proof with it when there is also an overt subject. Alternatively, the meaning constructor can be introduced only when there is no overt subject.

[3]Although there can be ambiguity even with valency information in case of verbs with several frames.

I	s	:	$E(\hat{*})$
persuaded	$\lambda x.\lambda y.\lambda P.\mathit{persuade}(x,y,P(y))$	:	$E(\hat{*}\ \textsc{subj}) \multimap E(\hat{*}\ \textsc{obj}) \multimap (E(\hat{*}\ \textsc{xcomp subj}) \multimap T(\hat{*}\ \textsc{xcomp})) \multimap T(\hat{*})$
him	a_1	:	$E(\hat{*})$
leave	$\lambda x.\mathit{admire}(x)$	:	$E(\hat{*}\ \textsc{subj}) \multimap T(\hat{*})$

Figure 6: Lexical entries for control infinitive

$$\dfrac{\dfrac{s : E(1) \qquad a_1 : E(3) \qquad \begin{array}{l}\lambda x.\lambda y.\lambda P.\mathit{persuade}(x,y,P(y)) : \\ \quad E(1) \multimap E(3) \multimap (E(6) \multimap T(5)) \multimap T(2)\end{array}}{\begin{array}{l}\lambda P.\mathit{persuade}(s,a_1,P(a_1)) : \\ \quad (E(6) \multimap T(5)) \multimap T(2)\end{array}} \qquad \begin{array}{l}\lambda x.\mathit{leave}(x) : \\ \quad E(6) \multimap T(5)\end{array}}{\mathit{persuade}(s,a_1,P(y)) \ : \ T(2)}$$

Figure 7: Proof for control infinitive structure

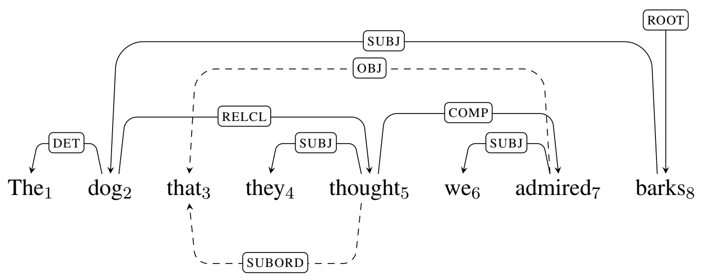

Figure 8: Two styles of relative clause annotation

they	a_1	:	$E(\hat{*})$
thought	$\lambda x.\lambda P.\mathit{think}(x,P)$	:	$E(\hat{*}\ \textsc{subj}) \multimap T(\hat{*}\ \textsc{comp}) \multimap T(\hat{*})$
we	s_+	:	$E(\hat{*})$
admired	$\lambda x.\lambda y.\mathit{admire}(x,y)$	:	$E(\hat{*}\ \textsc{subj}) \multimap E(\hat{*}\ \textsc{obj}) \multimap T(\hat{*})$
thought-RELCL	$\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$	:	$\forall \xi.(E(\xi) \multimap T(\hat{*})) \multimap (E(\uparrow) \multimap T(\uparrow)) \multimap E(\uparrow) \multimap T(\uparrow)$

Figure 9: Lexical entries for sample relative clause

$$\dfrac{\dfrac{\dfrac{\lambda y.\mathit{admire}(s_+,y) \ : \ E(9) \multimap T(7) \qquad [\mathbf{x_1} : E(9)]^1}{\mathit{admire}(s_+,\mathbf{x_1}) \ : \ T(7)} \qquad \lambda P.\mathit{think}(a_1,P) \ : \ T(7) \multimap T(5)}{\mathit{think}(a_1,\mathit{admire}(s_+,\mathbf{x_1})) \ : \ T(5)}}{\lambda x.\mathit{think}(a_1,\mathit{admire}(s_+,x)) \ : \ E(9) \multimap T(5)} \multimap_{\mathcal{I},1}$$

Figure 10: Proof structure for the relative clause

is the crucial part of the proof in Figure 10: abstraction over the object of *admire* gives us the set of x such that they think we admire x as the meaning of the relative clause. The next step is to intersect this meaning with the meaning of *dog* which has the meaning constructor $\lambda x.dog(x) : E(2) \multimap T(2)$. The meaning constructor **thought-RELCL** from Figure 9 will do this, though we do not show it in the proof.

What we call **thought-RELCL** is different from other meaning constructors in several ways. First, it is a *constructional* meaning, i.e. it is not associated with a lexical item alone, but triggered by a syntactic configuration, in this case a verb that bears the RELCL relation. To interpret the paths correctly, it must still be associated with a node in the tree, in this case naturally the verb of the relative clause. However, we cannot simply precombine **thought-RELCL** and the meaning of **thought** because the verb must combine with its arguments first. Thus, Glue Semantics allows us to split the meaning contributions at the interface to semantics without giving up on lexical integrity in the syntax.

Second, we see that this meaning constructor uses quantification over possible gaps, i.e. it requires some type E resource to be missing in the relative clause: of course, we only get a successful proof if this is instantiated to the introduced index 9 of the actually missing element, the object of *admire*. In this way, we can construct a proper meaning for relative clauses without a commitment to empty categories in the syntax. Another virtue of this approach is that we can restrict relativization sites if needed. In Figure 9 we use universal quantification, which allows all kinds of gaps, but as we mentioned above, it is also possible to use regular expressions to express non-deterministic paths. For example, if the language we analyze only allows e.g. relativization on local subjects and objects, we can use $E(\hat{*}\ (\textsc{subj}|\textsc{obj}))$, and if the language allows non-local relativization, but only on subjects and objects, we can use $E(\hat{*}\ \textsc{comp}^{*}\ (\textsc{subj}|\textsc{obj}))$. Whether such an interface restriction on relativization is preferrable to a purely syntactic one is an empirical question, but given the widespread avoidance of empty syntactic categories in Dependency Grammar, this approach at least offers an alternative.

6 Conclusion

We have seen how Glue Semantics lets us connect dependency syntax analyses to formal semantics by drawing on a framework that is quite close in spirit to Dependency Grammar, namely Lexical Functional Grammar. In particular, both frameworks reject the adoption of abstract syntactic analysis merely for the purpose of syntax-semantics homomorphism; and both frameworks assume lexical integrity and therefore reject decomposing lexical items in the syntax. Glue Semantics gives us fine-grained control over the syntax-semantics interface, allowing us to achieve the effects of empty categories and lexical decomposition there while preserving the surface-oriented syntactic analyses characteristic of both frameworks.

The most immediate advantage for Dependency Grammar is that this opens the door to the large literature in formal semantics. We have seen how we can analyse quantifier scope, control infinitives, and constructions with gaps, such as relative clauses. This is valuable in itself and can of course be extended to numerous other constructions.

But there is a reason why the interface to semantics is particularly important to Dependency Grammar. One way of motivating the surface-oriented structures that are often adopted in Dependency Grammar is by delegating to semantics the work that abstract syntax does in other frameworks. But if this is to go beyond mere hand-waving, it is important to accompany such claims with explicit analysis. We hope this paper has shown one way this can be done.

Acknowledgements

This research was funded by the Norwegian Research Council, project 300495 *Universal Natural Language Understanding*.

References

Ash Asudeh. 2012. *The logic of pronominal resumption*. Oxford University Press, Oxford.

Ash Asudeh. 2022. Glue Semantics. *Annual Review of Linguistics*, 8:321–341.

Laura Banarescu, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer, and Nathan Schneider. 2013. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th Linguistic Annotation Workshop and Interoperability with Discourse*, pages 178–186, Sofia, Bulgaria. Association for Computational Linguistics.

Joan Bresnan and Sam A Mchombo. 1995. The lexical integrity principle: Evidence from bantu. *Natural Language & Linguistic Theory*, 13(2):181–254.

Ann Copestake, Dan Flickinger, Carl Pollard, and Ivan A Sag. 2005. Minimal recursion semantics: An introduction. *Research on language and computation*, 3(2):281–332.

Richard Crouch and Aikaterini-Lida Kalouli. 2018. Named graphs for semantic representation. In *Proceedings of the Seventh Joint Conference on Lexical and Computational Semantics*, pages 113–118, New Orleans, Louisiana. Association for Computational Linguistics.

Haskell B. Curry and Robert Feys. 1958. *Combinatory logic: volume I.* North Holland, Amsterdam.

Mary Dalrymple, John Lamping, Fernando Pereira, and Vijay Saraswat. 1999. Overview and introduction. In Mary Dalrymple, editor, *Semantics and syntax in Lexical Functional Grammar: the resource logic approach*. MIT Press, Cambridge, MA.

Mary Dalrymple, John Lamping, and Vijay Saraswat. 1993. LFG semantics via constraints. In *Sixth Conference of the European Chapter of the Association for Computational Linguistics*, Utrecht, The Netherlands. Association for Computational Linguistics.

Mary Dalrymple, John J. Lowe, and Louise Mycock. 2019. *The Oxford Reference Guide to Lexical Functional Grammar*. Oxford University Press, Oxford.

Marie-Catherine de Marneffe, Christopher D. Manning, Joakim Nivre, and Daniel Zeman. 2021. Universal Dependencies. *Computational Linguistics*, 47(2):255–308.

B. Elan Dresher and Harry van der Hulst. 1998. Head–dependent asymmetries in phonology: complexity and visibility. *Phonology*, 15(3):317–352.

Jean-Yves Girard. 1987. Linear logic. *Theoretical Computer Science*, 50(1):1–102.

Matthew Gotham and Dag Trygve Truslew Haug. 2018. Glue semantics for universal dependencies. In *Proceedings of the LFG'18 Conference, University of Vienna*, pages 208–226, Stanford, CA. CSLI Publications.

Dag T. T. Haug. 2014. Partial Dynamic Semantics for Anaphora: Compositionality without Syntactic Coindexation. *Journal of Semantics*, 31(4):457–511.

Julia Hockenmaier and Mark Steedman. 2007. CCGbank: A corpus of CCG derivations and dependency structures extracted from the Penn Treebank. *Computational Linguistics*, 33(3):355–396.

William A. Howard. 1980. The formulae-as-types notion of construction. In *To H. B. Curry: essays on combinatory logic, lambda calculus, and formalism*, pages 479–490. Academic Press, London. Circulated in unpublished form from 1969.

Sylvain Kahane. 2003. The meaning-text theory. In *Dependency and Valency, Handbooks of Linguistics and Communication Sciences*. De Gruyter.

Sylvain Kahane. 2005. Structure des représentations logiques et interface sémantique-syntaxe. In *Actes de la 12ème conférence sur le Traitement Automatique des Langues Naturelles*, pages 153–162, Dourdan, France. Association pour le Traitement Automatique des Langues.

Aikaterini-Lida Kalouli and Richard Crouch. 2018. GKR: the graphical knowledge representation for semantic parsing. In *Proceedings of the Workshop on Computational Semantics beyond Events and Roles*, pages 27–37, New Orleans, Louisiana. Association for Computational Linguistics.

Hans Kamp and Uwe Reyle. 1993. *From Discourse to Logic: Introduction to Modeltheoretic Semantics of Natural Language, Formal Logic and Discourse Representation Theory*. Dordrecht: Kluwer Academic Publishers.

Ronald M. Kaplan and Joan Bresnan. 1982. Lexical-Functional Grammar: a formal system for grammatical representation. In Joan Bresnan, editor, *The mental representation of grammatical relations*, pages 173–281. MIT Press, Cambridge, MA.

Miltiadis Kokkonidis. 2007. First-order glue. *Journal of Logic, Language and Information*, 17:43–68.

Alexander Koller, Stephan Oepen, and Weiwei Sun. 2019. Graph-based meaning representations: Design and processing. In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics: Tutorial Abstracts*, pages 6–11, Florence, Italy. Association for Computational Linguistics.

Richard Montague. 1973. The proper treatment of quantification in ordinary English. In K. Jaakko, J. Hintikka, Julius M. E. Moravcsik, and Patrick Suppes, editors, *Approaches to natural language: proceedings of the 1970 Stanford workshop on grammar and semantics*, number 49 in Synthese Library, pages 221–243. Reidel, Dordrecht.

Siva Reddy, Oscar Täckström, Slav Petrov, Mark Steedman, and Mirella Lapata. 2017. Universal semantic parsing. In *Proceedings of the 2017 Conference on Empirical Methods in Natural Language Processing*, pages 89–101, Copenhagen, Denmark. Association for Computational Linguistics.

Greg Restall. 2000. *An introduction to substructural logics*. Routledge, London.

Livio Robaldo. 2006. *Dependency Tree Semantics*. Ph.D. thesis, University of Turin.

Petr Sgall, Eva Hajičová, and Jarmila Panevová. 1986. *The meaning of the sentence in its semantic and pragmatic aspects*. Academia, Prague.

Daniel Zeman and Jan Hajic. 2020. FGD at MRP 2020: Prague tectogrammatical graphs. In *Proceedings of the CoNLL 2020 Shared Task: Cross-Framework Meaning Representation Parsing*, pages 33–39, Online. Association for Computational Linguistics.