Formal Semantics for Dependency Grammar

Dag Trygve Truslew Haug and Jamie Yates Findlay

University of Oslo

Georgetown University Round Table
10 March 2023

Semantics for Dependency Grammar

- Dependency Syntax analyses sentence structure in terms of binary, asymmetric relations between words (dependencies)

Semantics for Dependency Grammar

- Dependency Syntax analyses sentence structure in terms of binary, asymmetric relations between words (dependencies)
- Typically adopts strong lexical integrity (Bresnan & Mchombo, 1995): words are the *only* atoms of syntax

Semantics for Dependency Grammar

- Dependency Syntax analyses sentence structure in terms of binary, asymmetric relations between words (dependencies)
- Typically adopts strong lexical integrity (Bresnan & Mchombo, 1995): words are the *only* atoms of syntax
- In principle agnostic about semantics, but with an affinity for graph-based formalisms
    - Tectogrammatical layer of FGD (Sgall et al., 1986)
    - Meaning-Text Theory (Mel'cuk et al., 1988; Kahane, 2003)
    - Abstract Meaning Representation (Banarescu et al., 2013)

Formal Semantics for Dependency Grammar

- Here we explore the possibility of connecting dependency grammar representations to formal (i.e. logic-based) semantics

Formal Semantics for Dependency Grammar

- Here we explore the possibility of connecting dependency grammar representations to formal (i.e. logic-based) semantics
- Two advantages:
  - Compositionality: 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

Formal Semantics for Dependency Grammar

- Here we explore the possibility of connecting dependency grammar representations to formal (i.e. logic-based) semantics
- Two advantages:
  - Compositionality: 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
  - Proof theory: answer questions like "if a set of sentences P is true, does it follow that sentence h is true?"

Formal Semantics for Dependency Grammar

- Here we explore the possibility of connecting dependency grammar representations to formal (i.e. logic-based) semantics
- Two advantages:
  - Compositionality: 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
  - Proof theory: answer questions like "if a set of sentences P is true, does it follow that sentence h is true?"
- Theoretical desiderata rather than necessarily the most practical way of creating semantic representation or tackle inference tasks

Formal Semantics for Dependency Grammar

- Here we explore the possibility of connecting dependency grammar representations to formal (i.e. logic-based) semantics
- Two advantages:
  - Compositionality: 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
  - Proof theory: answer questions like "if a set of sentences P is true, does it follow that sentence h is true?"
- Theoretical desiderata rather than necessarily the most practical way of creating semantic representation or tackle inference tasks
- Also, practical benefit of connecting to a rich branch of ongoing work

Frege's principle

- Logic trivially solves the proof theory requirement, but what about compositionality?
- Frege's principle: 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

Frege's principle

- Logic trivially solves the proof theory requirement, but what about compositionality?
- Frege's principle: 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 version of compositionality than we required!
- Not trivial to build logical representations from natural language compositionality

The basic idea

(1) a. Every man loves Chris

b. $\forall x.man(x) \rightarrow love(x, c)$

The basic idea

(1) a. Every man loves Chris

b. $\forall x.man(x) \rightarrow love(x, c)$

every	???
man	*man*
loves	*love*
Chris	*c*

The basic idea

(1) a. Every man loves Chris
b. $\forall x.man(x) \rightarrow love(x, c)$

every	$\forall x.? \rightarrow ?$
man	*man*
loves	*love*
Chris	*c*

The basic idea

(1) a. Every man loves Chris

b. $\forall x.man(x) \rightarrow love(x, c)$

every	$\lambda P.\lambda Q.\forall x.P(x) \rightarrow Q(x)$
man	*man*
loves	*love*
Chris	c

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning
  - $\rightarrow$ any non-lexical ambiguity must be syntactic

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning
  - $\rightarrow$ any non-lexical ambiguity must be syntactic
  - lambda calculus requires binary branching trees

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning
  - $\rightarrow$ any non-lexical ambiguity must be syntactic
  - lambda calculus requires binary branching trees
  - dependency trees (as in UDepLambda, Reddy et al. 2017) can be binarized but this freezes scope relations

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning
  - $\rightarrow$ any non-lexical ambiguity must be syntactic
  - lambda calculus requires binary branching trees
  - dependency trees (as in UDepLambda, Reddy et al. 2017) can be binarized but this freezes scope relations
- Lexical integrity:
  - single lexical items might provide different meanings that interact with other elements of the sentence in complex ways

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning
  - $\rightarrow$ any non-lexical ambiguity must be syntactic
  - lambda calculus requires binary branching trees
  - dependency trees (as in UDepLambda, Reddy et al. 2017) can be binarized but this freezes scope relations
- Lexical integrity:
  - single lexical items might provide different meanings that interact with other elements of the sentence in complex ways
  - finite verbs might introduce predicate-argument structure, temporal meaning *and* modal meaning

Combining Dependency grammar and Montague grammar

- The homomorphism problem:
  - strict compositionality requires that syntax and lexicon jointly *determine* meaning
  - $\rightarrow$ any non-lexical ambiguity must be syntactic
  - lambda calculus requires binary branching trees
  - dependency trees (as in UDepLambda, Reddy et al. 2017) can be binarized but this freezes scope relations
- Lexical integrity:
  - single lexical items might provide different meanings that interact with other elements of the sentence in complex ways
  - finite verbs might introduce predicate-argument structure, temporal meaning *and* modal meaning
  - $\rightarrow$ meanings might need to be scattered around the composition tree (cf. abstract heads in Chomskyan syntax)

Glue semantics

We will solve these problems using ideas from Glue Semantics (Dalrymple et al., 1993; Asudeh, 2022), which was developed within the tradition of another lexicalist theory of syntax that also does not enforce binary syntax, namely Lexical Functional Grammar (Kaplan & Bresnan, 1982; Dalrymple et al., 2019)

Glue semantics

We will solve these problems using ideas from Glue Semantics (Dalrymple et al., 1993; Asudeh, 2022), which was developed within the tradition of another lexicalist theory of syntax that also does not enforce binary syntax, namely Lexical Functional Grammar (Kaplan & Bresnan, 1982; Dalrymple et al., 2019)

- Alread applied to other frameworks: HPSG (Asudeh & Crouch, 2002), LTAG (Frank & van Genabith, 2001) and Minimalism (Gotham, 2018).

Glue semantics

We will solve these problems using ideas from Glue Semantics (Dalrymple et al., 1993; Asudeh, 2022), which was developed within the tradition of another lexicalist theory of syntax that also does not enforce binary syntax, namely Lexical Functional Grammar (Kaplan & Bresnan, 1982; Dalrymple et al., 2019)

- Alread applied to other frameworks: HPSG (Asudeh & Crouch, 2002), LTAG (Frank & van Genabith, 2001) and Minimalism (Gotham, 2018).
- Basically involves using a *composition logic*

Glue semantics

We will solve these problems using ideas from Glue Semantics (Dalrymple et al., 1993; Asudeh, 2022), which was developed within the tradition of another lexicalist theory of syntax that also does not enforce binary syntax, namely Lexical Functional Grammar (Kaplan & Bresnan, 1982; Dalrymple et al., 2019)

- Alread applied to other frameworks: HPSG (Asudeh & Crouch, 2002), LTAG (Frank & van Genabith, 2001) and Minimalism (Gotham, 2018).
- Basically involves using a *composition logic*

  A crude characterisation would be that glue semantics is like categorial grammar and its semantics, but without the categorial grammar.

(Crouch & van Genabith, 2000, 91)

The idea behind Glue semantics

The syntax is not (and should not be!) specific enough to guide composition of lambda terms, so lets introduce a specific combinatory logic for lambda terms:

The idea behind Glue semantics

The syntax is not (and should not be!) specific enough to guide composition of lambda terms, so lets introduce a specific combinatory logic for lambda terms:

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

The idea behind Glue semantics

The syntax is not (and should not be!) specific enough to guide composition of lambda terms, so lets introduce a specific combinatory logic for lambda terms:

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

- This looks a lot like categorial grammar!

The idea behind Glue semantics

The syntax is not (and should not be!) specific enough to guide composition of lambda terms, so lets introduce a specific combinatory logic for lambda terms:

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

- This looks a lot like categorial grammar!
- But decoupled from surface syntax, which is good for universality

The idea behind Glue semantics

The syntax is not (and should not be!) specific enough to guide composition of lambda terms, so lets introduce a specific combinatory logic for lambda terms:

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

- This looks a lot like categorial grammar!
- But decoupled from surface syntax, which is good for universality
- By making *love* type $A \multimap B \multimap C$, did we just stipulate a particular composition order? No!

Connecting up with the syntax

- The lexical entry of *love* cannot really be based on atomic categories like A, B and C.

Connecting up with the syntax

- The lexical entry of *love* cannot really be based on atomic categories like A, B and C.
- Instead, we will use a first-order system where predicates are *type constructors* that apply to nodes as terms to yield *types*

Connecting up with the syntax

- The lexical entry of *love* cannot really be based on atomic categories like A, B and C.
- Instead, we will use a first-order system where predicates are *type constructors* that apply to nodes as terms to yield *types*
- Writing $(\hat{*})$ for the current node
  - $E(\hat{*})$ is a type e meaning for that node
  - $T(\hat{*})$ is a type t meaning for that node
  - $E(\hat{*}) \multimap T(\hat{*})$ is a function type between the two (e.g. the type of a bare noun)

Connecting up with the syntax

- The lexical entry of *love* cannot really be based on atomic categories like A, B and C.
- Instead, we will use a first-order system where predicates are *type constructors* that apply to nodes as terms to yield *types*
- Writing $(\hat{*})$ for the current node
  - $E(\hat{*})$ is a type e meaning for that node
  - $T(\hat{*})$ is a type t meaning for that node
  - $E(\hat{*}) \multimap T(\hat{*})$ is a function type between the two (e.g. the type of a bare noun)
- Morever, we can use *path descriptions* based on syntactic labels (and $\uparrow$ for the mother node)
  - $E(\hat{*}\ \textsc{subj})$ is a type e meaning for the current node's subject

Connecting up with the syntax

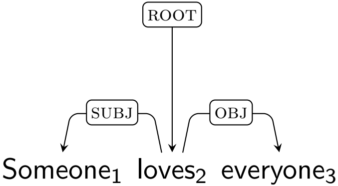
- The lexical entry of *love* cannot really be based on atomic categories like A, B and C.
- Instead, we will use a first-order system where predicates are *type constructors* that apply to nodes as terms to yield *types*
- Writing $(\hat{*})$ for the current node
  - $E(\hat{*})$ is a type e meaning for that node
  - $T(\hat{*})$ is a type t meaning for that node
  - $E(\hat{*}) \multimap T(\hat{*})$ is a function type between the two (e.g. the type of a bare noun)
- Morever, we can use *path descriptions* based on syntactic labels (and $\uparrow$ for the mother node)
  - $E(\hat{*}\ \textsc{subj})$ is a type e meaning for the current node's subject
  - $E(\hat{*}\ \textsc{obj})$ is a type e meaning for the current node's object

Connecting up with the syntax

- The lexical entry of *love* cannot really be based on atomic categories like A, B and C.
- Instead, we will use a first-order system where predicates are *type constructors* that apply to nodes as terms to yield *types*
- Writing $(\hat{*})$ for the current node
  - $E(\hat{*})$ is a type e meaning for that node
  - $T(\hat{*})$ is a type t meaning for that node
  - $E(\hat{*}) \multimap T(\hat{*})$ is a function type between the two (e.g. the type of a bare noun)
- Morever, we can use *path descriptions* based on syntactic labels (and $\uparrow$ for the mother node)
  - $E(\hat{*}\ \textsc{subj})$ is a type e meaning for the current node's subject
  - $E(\hat{*}\ \textsc{obj})$ is a type e meaning for the current node's object
  - $E(\hat{*}\ \textsc{subj}) \multimap E(\hat{*}\ \textsc{obj}) \multimap T(\hat{*})$ is the type we need for *love*

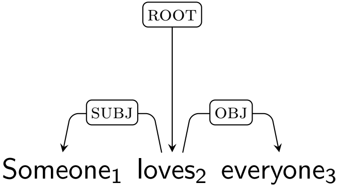
Examples

Simple transitive



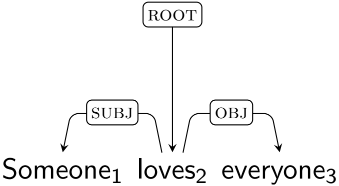
- We type
  - *someone* as $(E(\hat{*}) \multimap T(\uparrow)) \multimap T(\uparrow)$, i.e. $(E(1) \multimap T(2)) \multimap T(2)$

Simple transitive



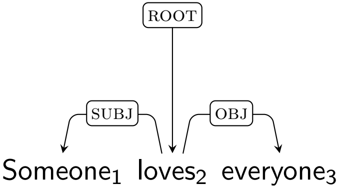
- We type
  - *someone* as $(E(\hat{\ast}) \multimap T(\uparrow)) \multimap T(\uparrow)$, i.e. $(E(1) \multimap T(2)) \multimap T(2)$
  - *love* $E(\hat{\ast}\ \textsc{subj}) \multimap E(\hat{\ast}\ \textsc{obj}) \multimap T(\hat{\ast})$, i.e. $E(1) \multimap E(3) \multimap T(2)$

Simple transitive



- We type
  - *someone* as $(E(\hat{\circledast}) \multimap T(\uparrow)) \multimap T(\uparrow)$, i.e. $(E(1) \multimap T(2)) \multimap T(2)$
  - *love* $E(\hat{\circledast}\ \textsc{subj}) \multimap E(\hat{\circledast}\ \textsc{obj}) \multimap T(\hat{\circledast})$, i.e. $E(1) \multimap E(3) \multimap T(2)$
  - *everyone* as $(E(\hat{\circledast}) \multimap T(\uparrow)) \multimap T(\uparrow)$, i.e. $(E(3) \multimap T(2)) \multimap T(2)$

Simple transitive



- We type
  - *someone* as $(E(\hat{*}) \multimap T(\uparrow)) \multimap T(\uparrow)$, i.e. $(E(1) \multimap T(2)) \multimap T(2)$
  - *love* $E(\hat{*}\ \textsc{subj}) \multimap E(\hat{*}\ \textsc{obj}) \multimap T(\hat{*})$, i.e. $E(1) \multimap E(3) \multimap T(2)$
  - *everyone* as $(E(\hat{*}) \multimap T(\uparrow)) \multimap T(\uparrow)$, i.e. $(E(3) \multimap T(2)) \multimap T(2)$
- This is isomorphic to the atomic types we used: $E(1) \mapsto A$, $E(3) \mapsto B$, $T(2) \mapsto C$ and so we get the same proofs

Prodrop

- Sometimes we neeed assume that positions that are not filled syntactically are nevertheless active in the semantics

Prodrop

- Sometimes we neeed assume that positions that are not filled syntactically are nevertheless active in the semantics
- For that, we adopt a *constructive* interpretation of Glue types

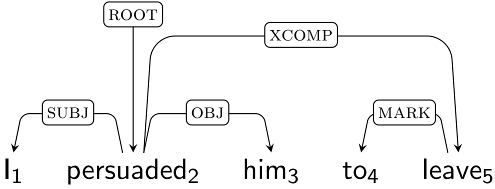
Prodrop

- Sometimes we neeed assume that positions that are not filled syntactically are nevertheless active in the semantics
- For that, we adopt a *constructive* interpretation of Glue types
- So if a verb type refers to its subject with $E(\hat{*}\ \textsc{subj})$ it *constructs* that semantic type

Prodrop

- Sometimes we neeed assume that positions that are not filled syntactically are nevertheless active in the semantics
- For that, we adopt a *constructive* interpretation of Glue types
- So if a verb type refers to its subject with $E(\hat{*}\ \mathrm{SUBJ})$ it *constructs* that semantic type
- We can then deal with prodrop using optional meaning constructors

Control infinitive



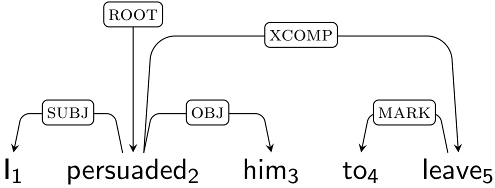
I $s : E(\hat{*})$

persuaded $\lambda x.\lambda y.\lambda P.persuade(x, y, P(y)) :$
$E(\hat{*}\ \textsc{subj}) \multimap E(\hat{*}\ \textsc{obj}) \multimap (E(\hat{*}\ \textsc{xcomp}\ \textsc{subj}) \multimap T(\hat{*}\ \textsc{xcomp})) \multimap T(\hat{*})$

him $a_1 : E(\hat{*})$

leave $\lambda x.admire(x) : E(\hat{*}\ \textsc{subj}) \multimap T(\hat{*})$

Control infinitive



I $s : E(1)$

persuaded $\lambda x.\lambda y.\lambda P.persuade(x, y, P(y)) :$
$E(1) \multimap E(3) \multimap (E(6) \multimap T(5)) \multimap T(2)$

him $a_1 : E(3)$

leave $\lambda x.admire(x) : E(6) \multimap T(5)$

Control infinitive

I $s : E(1)$

persuaded $\lambda x.\lambda y.\lambda P.persuade(x, y, P(y)) :$
$E(1) \multimap E(3) \multimap (E(6) \multimap T(5)) \multimap T(2)$

him $a_1 : E(3)$

leave $\lambda x.admire(x) : E(6) \multimap T(5)$

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

Control infinitive

I $s : E(1)$

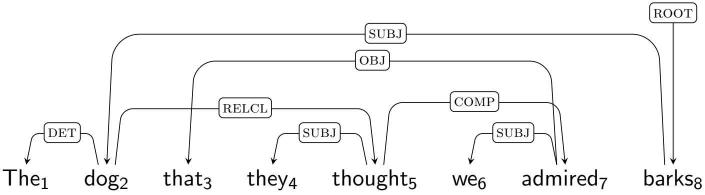
persuaded $\lambda x.\lambda y.\lambda P.persuade(x, y, P(y)) :$
$E(1) \multimap E(3) \multimap (E(6) \multimap T(5)) \multimap T(2)$

him $a_1 : E(3)$

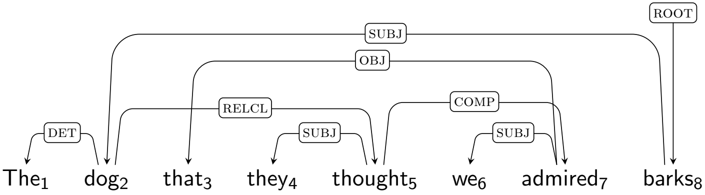
leave $\lambda x.admire(x) : E(6) \multimap T(5)$

$$\frac{\lambda P.persuade(s, a_1, P(a_1)) : (E(6) \multimap T(5)) \multimap T(2) \qquad \lambda x.leave(x) : E(6) \multimap T(5)}{persuade(s, a_1, leave(a_1)) : T(2)}$$

Relative clauses

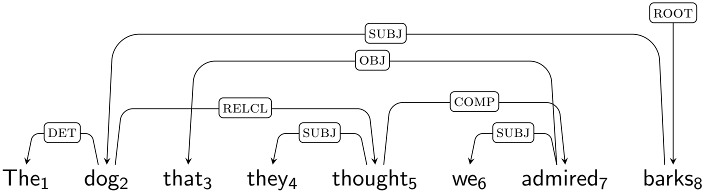


Relative clauses



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

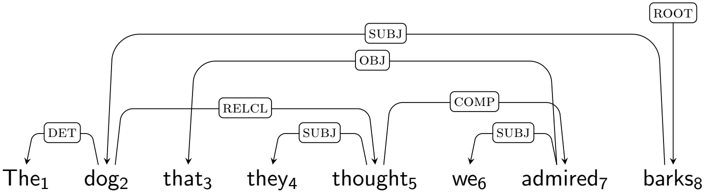
Relative clauses



they	a_1	:	$E(4)$
thought	$\lambda x.\lambda P.think(x, P)$	:	$E(4) \multimap T(7) \multimap T(5)$
we	s_+	:	$E(6)$
admired	$\lambda x.\lambda y.admire(x, y)$	:	$E(6) \multimap E(3) \multimap T(7)$
thought-RELCL	$\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$	:	$(E(3) \multimap T(5)) \multimap (E(2) \multimap T(2)) \multimap E(2) \multimap T(2)$

- The gap is identified with the object position of *admire*

Relative clauses



they	a_1	:
thought	$\lambda x.\lambda P.think(x, P)$	:
we	s_+	:
admired	$\lambda x.\lambda y.admire(x, y)$	:
thought-RELCL	$\lambda P.\lambda Q.\lambda x.P(x) \wedge Q(x)$	:

- The gap is identified with the object position of *admire*
- **thought-RELCL** allows for *any* gap position ($\forall \xi$)
- We can restrict the gap position at the syntax-semantics interface

Restricted indeterminacy

- 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.

Restricted indeterminacy

- 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.
- Node-references can be non-deterministic:
  - $\hat{*}$ $\textsc{obj}$ = the object daughter
  - $\hat{*}$ $(\textsc{subj}|\textsc{obj})$ = the set of the subject and object daughter,
  - $\hat{*}$ $\textsc{comp}^{*}$ $\textsc{subj}$ = the set of $\textsc{subj}$ daughters embedded under zero or more $\textsc{comp}$ daughters.
- Useful if you don't think gaps exist in the syntax!

Conclusions

- Glue allows us to build logical representations off dependency trees without requiring arbitrary binarization or lexical decomposition

Conclusions

- Glue allows us to build logical representations off dependency trees without requiring arbitrary binarization or lexical decomposition
- The syntax can underspecify the semantics, making it easier to "offload" work to the interface

Conclusions

- Glue allows us to build logical representations off dependency trees without requiring arbitrary binarization or lexical decomposition
- The syntax can underspecify the semantics, making it easier to "offload" work to the interface
- Opens an avenue for connecting to a rich strand of work in semantics
- For a concrete implementation based on this system, listen to our talk tomorrow!

References I

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

Asudeh, Ash & Richard Crouch. 2002. Glue Semantics for HPSG. In Frank van Eynde, Lars Hellan & Dorothee Beermann (eds.), *Proceedings of the 8th international HPSG conference*, Stanford, CA: CSLI Publications.

Banarescu, Laura, Claire Bonial, Shu Cai, Madalina Georgescu, Kira Griffitt, Ulf Hermjakob, Kevin Knight, Philipp Koehn, Martha Palmer & Nathan Schneider. 2013. Abstract Meaning Representation for sembanking. In *Proceedings of the 7th linguistic annotation workshop and interoperability with discourse*, 178–186. Sofia, Bulgaria: Association for Computational Linguistics. https://aclanthology.org/W13-2322.

References II

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

Crouch, Richard & Josef van Genabith. 2000. Linear logic for linguists. ESSLLI 2000 course notes.

Dalrymple, Mary, John Lamping & 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. https://aclanthology.org/E93-1013.

Dalrymple, Mary, John J. Lowe & Louise Mycock. 2019. *The Oxford reference guide to Lexical Functional Grammar*. Oxford: Oxford University Press.

References III

Frank, Anette & Josef van Genabith. 2001. GlueTag: Linear logic-based semantics for LTAG—and what it teaches us about LFG and LTAG. In Miriam Butt & Tracy Holloway King (eds.), *Proceedings of the LFG01 conference*, Stanford, CA: CSLI Publications.

Gotham, Matthew. 2018. Making Logical Form type-logical. *Linguistics and Philosophy* doi:10.1007/s10988-018-9229-z. In press.

Kahane, Sylvain. 2003. The meaning-text theory. In *Dependency and valency, handbooks of linguistics and communication sciences*, De Gruyter.

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

References IV

Mel'cuk, Igor Aleksandrovic et al. 1988. *Dependency syntax: theory and practice*. Albany: SUNY press.

Reddy, Siva, Oscar Täckström, Slav Petrov, Mark Steedman & Mirella Lapata. 2017. Universal semantic parsing. In *Proceedings of the 2017 conference on empirical methods in natural language processing*, 89–101. Copenhagen, Denmark: Association for Computational Linguistics. doi:10.18653/v1/D17-1009. https://aclanthology.org/D17-1009.

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

Inference rules

Application : implication elimination

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

Inference rules

Application : implication elimination

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

Abstraction : implication introduction

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

Switching argument orders

$$\dfrac{\dfrac{\dfrac{\lambda x.\lambda y.love(x,y) : A \multimap B \multimap C \qquad [\mathrm{n} : A]^1}{\lambda y.love(\mathrm{n},y) : B \multimap C} \qquad [\mathrm{j} : B]^2}{\dfrac{love(n,j) : C}{\lambda x.love(x,\mathrm{n}) : A \multimap C} \multimap_{\mathcal{I},1}}}{\lambda y.\lambda x.love(x,y) : B \multimap A \multimap C} \multimap_{\mathcal{I},2}$$

- $(A \multimap B \multimap C) \multimap (B \multimap A \multimap C)$ is a theorem of linear logic

Switching argument orders

$$
\dfrac{\dfrac{\dfrac{\dfrac{\lambda x.\lambda y.\mathit{love}(x,y) : A \multimap B \multimap C \qquad [\mathrm{n} : A]^1}{\lambda y.\mathit{love}(\mathrm{n},y) : B \multimap C} \qquad [\mathrm{j} : B]^2}{\mathit{love}(n,j) : C}}{\lambda x.\mathit{love}(x,\mathrm{n}) : A \multimap C}\multimap_{\mathcal{I},1}}{\lambda y.\lambda x.\mathit{love}(x,y) : B \multimap A \multimap C}\multimap_{\mathcal{I},2}
$$

- $(A \multimap B \multimap C) \multimap (B \multimap A \multimap C)$ is a theorem of linear logic
- There is also other stuff we get for free, like type raising operators $(A \multimap ((A \multimap B) \multimap B))$

Switching argument orders

$$\dfrac{\dfrac{\dfrac{\dfrac{\lambda x.\lambda y.love(x,y) : A \multimap B \multimap C \qquad [\mathrm{n} : A]^1}{\lambda y.love(\mathrm{n},y) : B \multimap C} \qquad [\mathrm{j} : B]^2}{love(n,j) : C}}{\lambda x.love(x,\mathrm{n}) : A \multimap C}\multimap_{\mathcal{I},1}}{\lambda y.\lambda x.love(x,y) : B \multimap A \multimap C}\multimap_{\mathcal{I},2}$$

- $(A \multimap B \multimap C) \multimap (B \multimap A \multimap C)$ is a theorem of linear logic
- There is also other stuff we get for free, like type raising operators $(A \multimap ((A \multimap B) \multimap B))$
- Moreover, there is an efficient proof algorithm (based on CYK) and a good implementation (Glue semantics workbench, (**?**))

Quantifier scope ambiguity: surface scope

$$
\dfrac{\dfrac{\dfrac{\dfrac{\boxed{\begin{array}{l}\textbf{loves}\\ \lambda x.\lambda y.love(x,y) :\\ A \multimap B \multimap C\end{array}} \qquad [\mathrm{x}_1 : A]^1}{\begin{array}{l}\lambda y.love(\mathrm{x}_1,y) :\\ B \multimap C\end{array}} \qquad \boxed{\begin{array}{l}\textbf{everyone}\\ \lambda P.\forall z.person(z) \to P(z) :\\ (B \multimap C) \multimap C\end{array}}}{\begin{array}{l}\forall z.person(z) \to love(\mathrm{x}_1,z) :\\ C\end{array}}}{\begin{array}{l}\lambda x.\forall z.person(z) \to love(x,z) :\\ A \multimap C\end{array}} \multimap_{\mathcal{I},1} \qquad \boxed{\begin{array}{l}\textbf{someone}\\ \lambda P.\exists x.person(x) \wedge P(x) :\\ (A \multimap C) \multimap C\end{array}}}{\begin{array}{l}\exists x.person(x) \wedge (\forall z.person(z) \to love(z,x)) :\\ C\end{array}}
$$

Quantifier scope ambiguity: inverse scope

$$\dfrac{\boxed{\begin{array}{l}\textbf{loves}\\ \lambda x.\lambda y.love(x,y):\\ A \multimap B \multimap C\end{array}} \qquad [x_1 : A]^1}{\lambda y.love(a,y): B \multimap C}$$

$$\dfrac{\lambda y.love(a,y): B \multimap C \qquad [x_2 : B]^2}{\dfrac{love(x_1,x_2): C}{\lambda x.love(x,x_2): A \multimap C}\multimap_{\mathcal{I},1}}$$

$$\dfrac{\lambda x.love(x,x_2): A \multimap C \qquad \boxed{\begin{array}{l}\textbf{someone}\\ \lambda P.\exists x.person(x) \wedge P(x):\\ (A \multimap C) \multimap C\end{array}}}{\dfrac{\exists x.person(x) \wedge love(x,x_2): C}{\lambda y.\exists x.person(x) \wedge love(x,y): B \multimap C}\multimap_{\mathcal{I},2}}$$

$$\dfrac{\lambda y.\exists x.person(x) \wedge love(x,y): B \multimap C \qquad \boxed{\begin{array}{l}\textbf{everyone}\\ \lambda P.\forall z.person(z) \rightarrow P(z):\\ (B \multimap C) \multimap C\end{array}}}{\forall z.person(z) \rightarrow (\exists x.person(x) \wedge love(x,z)): C}$$