Computational Glue semantics

Mark-Matthias Zymla[1]
[1]University of Konstanz
mark-matthias.zymla@uni-konstanz.de

1 Day 1: Introduction

Computational Glue deals with implementational challenges provided by Glue semantics. The goal is to provide a system that allows researchers to faithfully replicate and confirm their analyses in a computational setting. However, broadly put, formal linguists can intuitively form comparison classes across proofs that are not easily discernible from a computational perspective.

1.1 Requirements

- Install the free version of Docker (`https://www.docker.com/products/docker-desktop/`;).
  - Docker will block some of the space on your hard drive, so make sure that you have enough space (ca. 5 gigabytes).
- Download and **unzip** the XLE+Glue project:
  `https://github.com/Mmaz1988/xleplusglue/archive/refs/heads/ESSLLI2025_glue_course.zip`

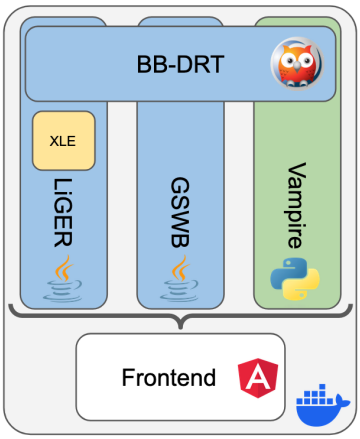


Figure 1: Why Docker?

1.2 Installation and usage

- The project contains a Stanza folder. Setup the Stanza folder as described in the `README.md` within the Stanza folder.
  - **IMPORTANT:** The Stanza folder contains `.sh` files that you can execute to automate the process (via terminal/other command line tool; more info in the `README.md`). There is one file for Windows users and one file for Unix users (which should run on both MacOS and LINUX).
- Start the Docker application (usually, Docker Desktop)
- Navigate to the Docker folder within the project using the terminal or other command line tool.
- Execute docker compose up –build in the terminal. Leave the terminal open. This might take some time, depending on Internet speed.
- Once this is done, you should be able to open UD+Glue at the address localhost:80 in a browser.
- There, you should be at least able to navigate to analysis (menu on the left) and analyze the default sentence. You should also be able to run the testsuite in regression testing by clicking the "Parse all" button under the settings.

1.3 Computational glue notation cheat sheet

The tools presented here use a notation that is supposed to be similar to theoretical notations to make them more accessible Not all computational Glue tools use the same notation

1.3.1 Linear logic

(1) **Some examples:**

a. `g_e`
b. `(g_e -o (d_e -o (i_s -o h_t)))`
c. `((i_s -o h_t) -o (t_s -o f_t))`
d. `AX_t.((d_e -o X_t) -o X_t)`
e. `((g_e -o g_t) -o AX_t.((d_e -o X_t) -o X_t))`
f. `AX_t.AY_s.((d_e -o (Y_s -o X_t)) -o(Y_s -o X_t))`

(2) `((g -o g) -o AX.((d -o X) -o X))`

(3) **Atomic elements:**

a. *Constants:* Strings of lower-case alphanumeric characters; optionally with type declaration

constant	**type**	
g	_e	= g_e

b. *Variables:* String of upper-case alphanumeric characters; optionally with type declaration

constant	**type**	
X	_t	= X_t

(4) **Linear logic formulas:**

a. *Linear implication:* (ϕ `-o` ψ), where ϕ, ψ are well-formed formulas
b. *Linear Quantification:* `AX.` ϕ, where ϕ is a well-formed formula

1.3.2 The meaning side

- The meaning side can contain arbitrary strings, but it also allows for the usage of typed lambda calculus or Prolog representations (for interfacing with other pieces of software)

(5) **Some examples of typed lambda calculus**

a. `[/x_e.sleep(x)]`
b. `[/x_e.[/w_s.sleep(x,w)]]`
c. `[/P_<e,t>.[/Q_<e,t>.[/x_e.(P(x) & Q(x))]]]`

(6) **Some examples of quantifiers:**

a. `Ex_e[dog(x) & bark(x)]`
b. `Ax_e[cat(x) -> sleep(x)]`
c. `[/P_<e,t>.[/Q_<e,t>.Ex_e[P(x) & Q(x)]]]`

(7) **Lambda terms and quantifier terms:**

	operator	*binder*	*type*	*scope*		
[	/	x	_e.	dog(x)	]	= `[/x_e.dog(x)]` $\equiv \lambda x_e.dog(x)$
	A	y	_e	[human(y)]		= `Ey_e[human(y)]` $\equiv \exists y_e[human(y)]$

- Note that lambda-expressions are wrapped in brackets or parentheses to indicate scope
- In quantifier expressions, the scope brackets (or parentheses) are wrapped around the actual scope

(8) **Operators:**

a. **Logical 'and' (&):** `(P(x) & Q(x))`
b. **Logical 'or'(v):** `(P(x) v Q(x))`
c. **Logical 'implication'(->):** `(P(x) -> Q(x))`
d. **Variant with brackets:** `Ex_e[P(x) & Q(x)]`
e. **Prefix notation:** `equals(x,y)`

(9) **Atomic types:** e, s, v, t

(10) **Complex types:**

a. `<e,<e,<s,t>>>`
b. `<<s,t>,<s,t>>`
c. `<<e,t>,t>`
d. `<<e,t>,<<e,t>,t>>`
e. `<<e,<s,t>>,<s,t>>`

2 Computational glue semantics: the basics

The resource-sensitivity hypothesis (Asudeh, 2004)

- Language is resource-sensitive
- ≈ Elements of a linguistic expression provide resources for producing meanings
- For simple examples, this can be conceptualized as a puzzle:

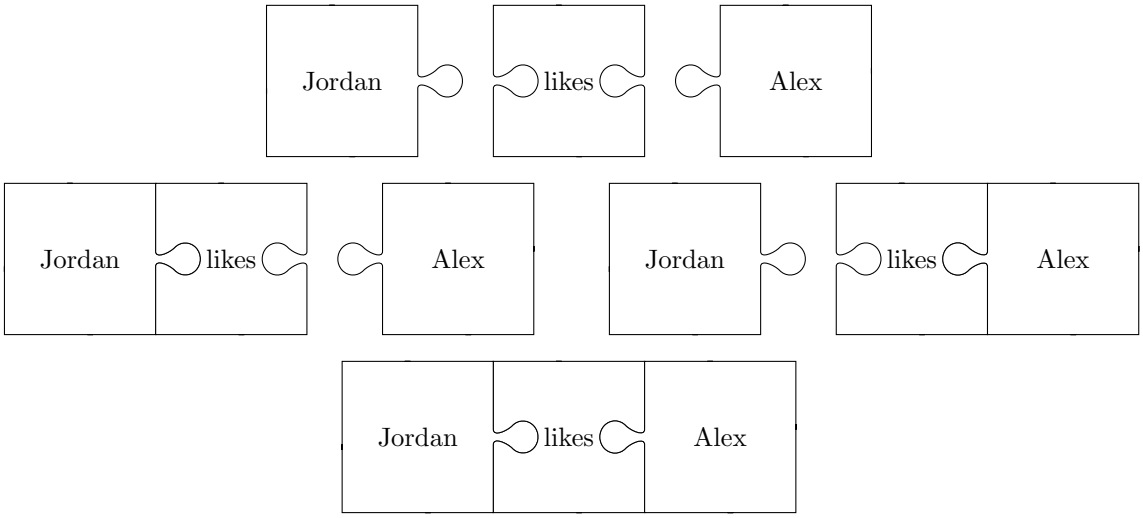


Figure 2: The resource-sensitivity hypothesis illustrated

$$\dfrac{\dfrac{g : Jordan \qquad g \multimap h \multimap f : \lambda x.\lambda y.like(x,y)}{h \multimap f : \lambda y.like(jordan,y)} \qquad h : Alex}{f : like(jordan, alex)}$$

Figure 3: Proof-tree for *Jordan likes Alex*

- The meaning language is isomorphic to the Glue language (the Curry-Howard isomorphism; CHI; Curry et al. 1958)

$$\dfrac{\begin{array}{c}[x : A]^i \\ \vdots \\ f(x) : B\end{array}}{\lambda x.f(x) : A \multimap B} \multimap_{I,i} \qquad \dfrac{f : A \multimap B \qquad a : A}{f(a) : B} \multimap_E$$

Figure 4: Implication introduction and elimination; CHI

→ The combinatoric possibilities are more complex than suggested by the simple puzzle examples

→ Still possible to convey the intuition using puzzle pieces

Exercise 1: Using the GSWB

(11) **TASK 1:**
The following examples demonstrate various examples with function application. Go through the examples and compare them. Does the computational implementation behave as expected?

```
{
love : (subj -o (obj -o sentence))
jordan : subj
sam : obj
}
{
[/x.[/y.love(x,y)]] : (g_e -o (h_e -o f_t))
jordan : g_e
sam : h_e
}
{
[/x_e.[/y_e.love(x,y)]] : (g_e -o (h_e -o f_t))
jordan : g_e
sam : h_e
}
{
[/y_e.[/x_e.love(x,y)]] : (h_e -o (g_e -o f_t))
jordan : g_e
sam : h_e
}
```

(12) **TASK 2:**

a. Produce meaning constructors for a proof with a **intransitive verb** and verify the proof with the Glue tool.

b. Produce meaning constructors for a proof with a **ditransitive verb** and verify the proof with the Glue tool.

The argument inversion example shows us that we can freely re-order the arguments of our verb as long as we do it for both the glue side and the meaning side of the verb meaning constructor. This is thanks to the Curry-Howard correspondence.

```
{
//Argument inversion
a : (h_e -o (g_e -o f_t))
a1 : ((g_e -o (h_e -o f_t)) -o goal_t)
}
```

(13) **QUESTION:**
Given that, how many solutions are possible for each of the first four meaning constructor sets above?

2.1 Semantic ambiguity

- Sometimes puzzle pieces with different meanings have the same shape

→ their order can be freely varied

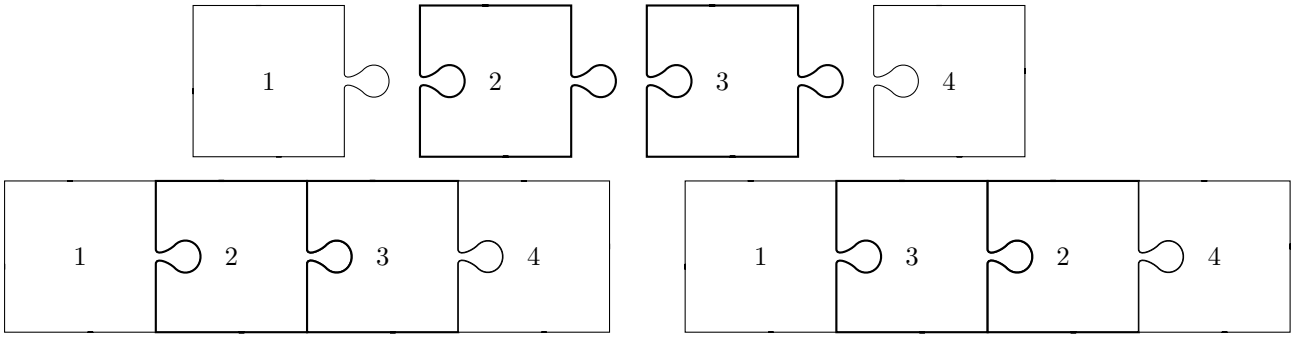


Figure 5: Ambiguity as puzzle

→ More complex issues related to ambiguity in Glue semantics can be broken down to the level of ordering pieces

Observation: Assuming a puzzle is a derivation, a derivation is successful if we can order the pieces in such a way that they all fit together, i.e., we enforce a possibly partial ordering on the input resources.

- In Glue semantics, this ordering is affected by two different aspects: the meaning side and the glue side of a meaning constructor
  - Linear logic determines a potentially partial order on combinations
  - A solution is one full ordering of combinations
  - A derivation may have multiple solutions
  - The meaning side determines which solutions are equivalent and which solutions are distinct (In Figure 5, if the meaning language is characterized by sets, then $s_1 = s_2$, but if its characterized by lists, then $s_1 \neq s_2$)
  - Some meaning languages may filter out additional solutions

2.1.1 Ambiguity arises from modifiers

- Intuitively, modifiers are a special type of meaning constructor that produces the same output as its input (see, e.g., Gupta and Lamping 1998; Lev 2007; cf. also Chung and Ladusaw 2003)

(14) a trustworthy scottish chairman

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

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

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

$$\dfrac{\lambda P.\lambda x.scottish(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t \qquad \dfrac{\lambda P.\lambda x.trustworthy(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t \quad \lambda x.chairman(x) : g_e \multimap g_t}{\lambda x.trustworthy(x) \wedge chairmen(x) : g_e \multimap g_t}}{\lambda x.scottish(x) \wedge trustworthy(x) \wedge chairmen(x) : g_e \multimap g_t}$$

Figure 6: Combinatory possibility 1

$$\dfrac{\lambda P.\lambda x.trustworthy(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t \qquad \dfrac{\lambda P.\lambda x.scottish(x) \wedge P(x) : (g_e \multimap g_t) \multimap g_e \multimap g_t \quad \lambda x.chairman(x) : g_e \multimap g_t}{\lambda x.scottish(x) \wedge chairmen(x) : g_e \multimap g_t}}{\lambda x.trustworthy(x) \wedge scottish(x) \wedge chairmen(x) : g_e \multimap g_t}$$

Figure 7: Combinatory possibility 2

- Modifiers are characterized by their linear logic side
- On the meaning side, they can be, e.g., partial identity functions, or, as we will see, quantifiers
- Whether the order of modifiers affects the semantic interpretation depends on the meaning side

2.2 Quantifiers – Two approaches

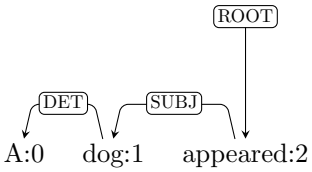
- **Quantification** over linear logic variables
- **Syntactic** assignment of scope landing sites
- As expected, for many cases these approaches make similar predictions

(15) $\lambda P_{et}.\exists x[dog(x) \wedge P(x)]$:

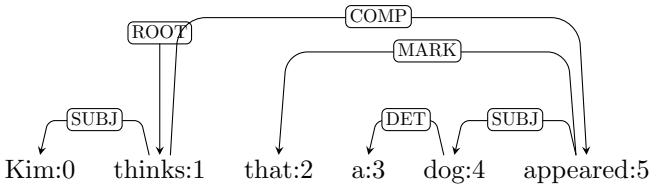
a.) $(\uparrow_e \multimap \%scope_t) \multimap \%scope_t$, where $\%scope$ is a dominating clausal structure (i.e., a structure yielding type t)

b.) $\forall X_t((\uparrow_e \multimap X_t) \multimap X_t)$, X is a variable over resources of type t

(16) **A dog appeared.**



Kim thinks that a dog appeared.



a. a dog $\Leftrightarrow$ $\forall X.(g \multimap X) \multimap X$ $\Leftrightarrow$ $(1 \multimap 2) \multimap 2$
b. a dog $\Leftrightarrow$ $\forall X.(4 \multimap X) \multimap X$ $\Leftrightarrow$ $(4 \multimap 5) \multimap 5$, $(4 \multimap 1) \multimap 1$

Here, we use the **positional encoding** of dependency trees to determine **resource indices**.

- Generalized quantifiers logically behave like restricted modifiers
- Generalized quantifiers are not pure modifiers; scope is constrained by an additional resource

(17) Every monkey likes a banana.

a. $\lambda x.\lambda y.\text{like}(x, y) : m_\sigma \multimap (b_\sigma \multimap f_\sigma)$
b. $\lambda P.\forall x[\text{monkey}(x) \rightarrow P(x)] : (m_\sigma \multimap f_\sigma) \multimap f_\sigma$
c. $\lambda Q.\exists y[\text{banana}(y) \wedge Q(y)] : (b_\sigma \multimap f_\sigma) \multimap f_\sigma$

$$\dfrac{\lambda P.\forall x[\text{monkey}(x) \rightarrow P(x)] : (m_e \multimap f_t) \multimap f_t \qquad \dfrac{\dfrac{[X : m_e]^1 \quad \lambda x.\lambda y.\text{like}(x, y) : m_e \multimap (b_e \multimap f_t)}{\lambda y.like(X, y) : b_e \multimap f_t}\multimap_E \qquad \dfrac{\lambda Q.\exists y[\text{banana}(y) \wedge Q(y)] : (b_e \multimap f_t) \multimap f_t}{\exists y[\text{banana}(y) \wedge \text{like}(X, y)] : f_t}\multimap_E}{\lambda x.\exists y[\text{banana}(y) \wedge \text{like}(x, y)] : m_e \multimap f_t}\multimap_{I,1}}{\forall x[\text{monkey}(x) \rightarrow \exists y[\text{banana}(y) \wedge \text{like}(x, y)]] : f_t}\multimap_E$$

Figure 8: Glue proof: *Every monkey likes a banana* surface scope

$$\dfrac{\lambda P.\forall x[\text{monkey}(x) \rightarrow P(x)] : (m_e \multimap f_t) \multimap f_t \qquad \dfrac{\dfrac{\dfrac{[X : m_e]^1 \quad \lambda x.\lambda y.\text{like}(x, y) : m_e \multimap (b_e \multimap f_t)}{\lambda y.\text{like}(X, y) : b_e \multimap f_t}\multimap_E \qquad [Y : b_e]^2}{\text{like}(X, Y) : f_t}\multimap_E}{\lambda x.\text{like}(x, Y) : m_e \multimap f_t}\multimap_{I,1}}{\dfrac{\forall x[\text{monkey}(x) \rightarrow \text{like}(x, Y)] : f_t}{\lambda y.\forall x[\text{monkey}(x) \rightarrow \text{like}(x, y)] : b_e \multimap f_t}\multimap_{I,2} \qquad \lambda Q.\exists y[\text{banana}(y) \wedge Q(y)] : (b_e \multimap f_t) \multimap f_t}\multimap_E$$
$$\exists y[\text{banana}(y) \wedge \forall x[\text{monkey}(x) \rightarrow \text{like}(x, y)]] : f_t$$

Figure 9: Glue proof: *Every monkey likes a banana* inverse scope

Exercise 2

(18) **TASK 1:**
Provide a proof for (14) and compare the output to the proofs in Figures 6 and 7.

(19) **TASK 2:**
Run the two proofs below. In this instance, they are equivalent. How could you change the second proof to enforce a specific order of the quantifiers? We are not necessarily looking for a sensible way, but for a straightforward way!

```
{
//Quantified implicational linear logic
[/P_<e,t>.Ax_e[dog(x) -> P(x)]] : AX_t.((d_e -o X_t) -o X_t)
[/P_<e,t>.Ax_e[cat(x) & P(x)]] : AY_t.((c_e -o Y_t) -o Y_t)
[/x_e.[/y_e.chase(x,y)]] : (d_e -o (c_e -o f_t))
}
{
//Implicational linear logic
[/P_<e,t>.Ax_e[dog(x) -> P(x)]] : ((d_e -o f_t) -o f_t)
[/P_<e,t>.Ax_e[cat(x) & P(x)]] : ((c_e -o f_t) -o f_t)
[/x_e.[/y_e.chase(x,y)]] : (d_e -o (c_e -o f_t))
}
```

(20) **QUESTION 1:**
What "problem" do you run into if you have, e.g., two existential quantifiers?

(21) **TASK 3:**

a. Extend your answer from (12-b) by replacing your simple arguments by quantifiers. Inspect the results: Do you get the right solutions?

(22) **QUESTION 2:** Intuitively, how do the provers deal with quantifier ambiguity?

2.3 A computational treatment of quantifiers

- **Observation:** The additional resource in the quantifier indicates a potential assumption
- Can we make this intuition more clear? Yes, by assigning polarities to linear logic formulas (borrowed from a particular way of deriving Glue proofs, proof nets; de Groote 1999)

→ Roughly, a glue derivation is possible if each positive assumption except the goal can be matched with a corresponding negative assumption

(23) $(e \multimap t) \multimap t,\ e \multimap t \vdash t$ e.g., *a dog barks*

$(e^- \multimap t^+)^+ \qquad ((e^+ \multimap t^-)^- \multimap t^+)^+$

- Higher-order linear logic compilation (Hepple, 1996; Lev, 2007) makes assumptions explicit
- **Compilation:** the process of decomposing complex antecedents in an LL formula into atomic antecedents such that it is fully right-associative, i.e., first-order formula (Hepple, 1996). Compiled formulas have charges which must be discharged by combining with (nested) assumptions.

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

b.
$$\dfrac{\dfrac{[e]^i \qquad e \multimap t}{t^i} \qquad t_i \multimap t}{\mathrm{t}}$$

- Assumptions provide a lower bound for the scoping of a generalized quantifier (cf. traces in Kratzer and Heim (1998)-style semantics)

→ The upper bound is determined by the quantification mechanism

2.4 Computational properties of quantifier ambiguity

- We have seen two approaches:
- **1) Implicational linear logic fragment (ILL)**
- **2) Quantificational implicational linear logic fragment (QILL)**

(25) $$\forall X_t.(X_t \multimap X_t) \quad + \quad \begin{matrix} a_t \\ b_t \\ c_t \end{matrix} \quad \Rightarrow \quad \begin{matrix} a_t \multimap a_t \\ b_t \multimap b_t \\ c_t \multimap c_t \end{matrix}$$

- QILL meaning constructors allow for a more economic notation of scope ambiguities

  but: they are equivalent to a set of sets of meaning constructors with all possible instantiations

- ILL with syntactic scope fixing potentially allows us to prune these sets (more on this later)

```
{                                  {                  {                  {
x : AX_t.(X_t -o X_t)              a : (a_t -o a_t)    a : (a_t -o a_t)    a : (a_t -o a_t)
x : AY_t.(Y_t -o Y_t)              a : (a_t -o a_t)    a : (a_t -o a_t)    a : (a_t -o a_t)
x : AZ_t.(Z_t -o Z_t)              b : (b_t -o b_t)    a : (a_t -o a_t)    a : (a_t -o a_t)
a : (a_t -o a_t)          ⇔    { b : (b_t -o b_t) }  , b : (b_t -o b_t) }  , b : (b_t -o b_t) }  , ... }
a : (a_t -o a_t)                   b : (b_t -o b_t)    b : (b_t -o b_t)    b : (b_t -o b_t)
b : (b_t -o b_t)                   b : (b_t -o b_t)    b : (b_t -o b_t)    b : (b_t -o b_t)
b : (b_t -o b_t)                   c : (a_t -o b_t)    c : (a_t -o b_t)    c : (a_t -o b_t)
c : (a_t -o b_t)                   d : a_t             d : a_t             d : a_t
d : a_t                            }                   }                   }
}
```

Figure 10: QILL vs. ILL example meaning constructors (see appendix B)

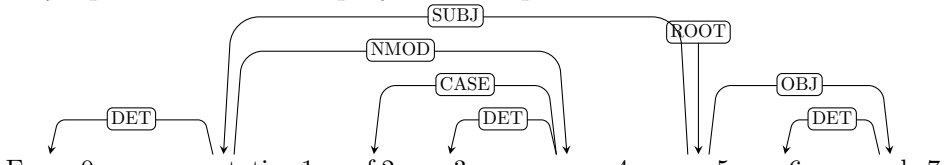
	QILL derivation	ILL derivation
No. of solutions	1344	1344
Duration	40ms	3ms
No. of combinations	6794	1955
No. of attempted combinations	34301	5216

Table 1: Evaluation metrics for chart-based and graph-based derivations of MCs in Figure 10

2.4.1 Nested quantifiers

(26) a. Every representative of a company saw a sample.

b.



c.

every representative	$\Leftrightarrow$	$\forall X.(g \multimap X) \multimap X$	$\Leftrightarrow$	$(1 \multimap 5) \multimap 5$
a sample	$\Leftrightarrow$	$\forall Z.(h \multimap Z) \multimap Z$	$\Leftrightarrow$	$(7 \multimap 5) \multimap 5$
a company	$\Leftrightarrow$	$\forall Y.(j \multimap Y) \multimap Y$	$\Leftrightarrow$	$(4 \multimap 2) \multimap 2, (4 \multimap 5) \multimap 5$

	QILL derivation	ILL derivation
No. of solutions	5	5
Duration	3ms	3ms
No. of combinations	35	15
No. of attempted combinations	183	$\varnothing$

Table 2: Evaluation metrics for chart-based and graph-based derivations of MCs in (26)

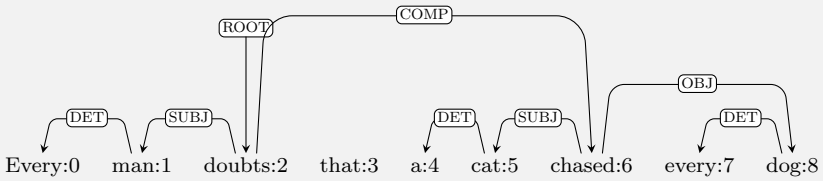
- Since quantifiers form a depencency with the predicates that they scope over, generally: **QILL $\approx$ ILL**
- QILL is much more computationally complex, but it is also more flexible

Exercise 3

(27) **TASK 1:**
Provide a proof for the following sentence using QILL:

a. Every man doubts that a cat chased (away) every dog.

b. Assume the following semantics for believe:
$\lambda P_t.\lambda x_e.doubt(x, P) : (6_t \multimap (1_e \multimap 2_t))$



(28) **QUESTION:**
How many meaning constructor sets in ILL does your QILL set correspond to?

(29) **TASK 2:**
Explain the commonalities and differences of the following proofs. What determines their complexity?

```
{
a : AV_t.((a_e -o V_t) -o V_t)
b : AW_t.((b_e -o W_t) -o W_t)
d : AX_t.((c_e -o X_t) -o X_t)
d : AY_t.((d_e -o Y_t) -o Y_t)
e : AZ_t.((e_e -o Z_t) -o Z_t)
order : (a_e -o (b_e -o (c_e -o (d_e -o (e_e -o f_t)))))
}
{
a : ((a_e -o f_t) -o f_t)
b : ((b_e -o f_t) -o f_t)
d : ((c_e -o f_t) -o f_t)
d : ((d_e -o f_t) -o f_t)
e : ((e_e -o f_t) -o f_t)
order : (a_e -o (b_e -o (c_e -o (d_e -o (e_e -o f_t)))))
}
{
a : (a -o a)
b : (a -o a)
c : (a -o a)
d : (a -o a)
e : (a -o a)
order : a
}
```

3 Day 2: Algorithms for handling Glue semantics

3.1 Chart-based glue semantics derivations (Hepple 1996,Lev 2007, ch. 5)

A **chart** is a data structure that stores partial solutions to re-use them later. Lots of early work in syntactic parsing is based on chart parsers. For us two concepts are important:

- The **agenda** stores elements that still need to be processed.
- The **chart** memorizes what already has been processed, avoiding repetition

The proof procedure in brief:

- Glue premises are compiled and indexed
- Initially all premises are put on an agenda and an empty chart is initialized

→ A premise is taken off the agenda

→ Iterate over the chart and attempt combinations

∗ For successful combination, index sets must be disjoint (ensures that each resource is used only once in a proof)

∗ When premises are combined their index sets are unified

→ Successful combinations are put back on the agenda

→ After all combinations have been computed, the current element is put on the chart

→ The algorithm terminates when the agenda is empty

3.1.1 A simple example

Agenda	**Chart**
$[0]Jordan : g$	
$[1]\lambda x.\lambda y.likes(x,y) : g \multimap h \multimap f$	
$[2]Alex : h$	

Agenda	**Chart**
$[1]\lambda x.\lambda y.likes(x,y) : g \multimap h \multimap f$	$[0]Jordan : g$
$[2]Alex : h$	

Agenda	**Chart**
$[2]Alex : h$	$[0]Jordan : g$
$[0,1]\lambda y.likes(jordan,y) : h \multimap f$	$[1]\lambda x.\lambda y.likes(x,y) : g \multimap h \multimap f$

Agenda	**Chart**
$[0,1]\lambda y.likes(jordan,y) : h \multimap f$	$[0]Jordan : g$
	$[1]\lambda x.\lambda y.likes(x,y) : g \multimap h \multimap f$
	$[2]Alex : h$

Agenda	**Chart**
$[0,1,2]likes(jordan,alex) : f$	$[0]Jordan : g$
	$[1]\lambda x.\lambda y.likes(x,y) : g \multimap h \multimap f$
	$[2]Alex : h$
	$[0,1]\lambda y.likes(jordan,y) : h \multimap f$

Agenda	**Chart**
$\varnothing$	$[0]Jordan : g$
	$[1]\lambda x.\lambda y.likes(x,y) : g \multimap h \multimap f$
	$[2]Alex : h$
	$[0,1]\lambda y.likes(jordan,y) : h \multimap f$
	$\mathbf{[0,1,2]likes(jordan,alex) : f}$

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

...

□

- Since resource usage is tracked by the index sets, each combination must be checked
- The algorithm also needs to keep track of assumptions and discharges during combination checks but the overall mechanism is the same

→ Overall simple algorithm; many unnecessary combination checks

3.2 Graph-based glue semantics derivations (Lev, 2007, ch. 6)

- Distinguishes between linear logic derivation and semantic derivation
- Linear logic derivations are category-based directed graph structures where each category in the input meaning constructors appears exactly once
- categories are connected via combination nodes such that any two matching category nodes $n_1 : A$ and $n_2 : A \multimap B$ lead into a combination node c_1, and c_1 leads into $n_3 : B$, the category resulting from their combination
- Sub-graphs that are strongly connected (SSCs) indicate that multiple solutions are possible (strongly connected subgraphs indicate a cycle in the derivation graph)
  - a graph is strongly connected if every vertex is reachable from every other vertex
  - To get all possible solutions for a SSC the chart algorithm is used
- Semantic derivations are read off of derviation graphs by tracing back combination histories
- Histories are actual data structures. They are associated with category nodes in the graph

Example proof for *Everyone saw someone*

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

$\longrightarrow_{compile}$

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

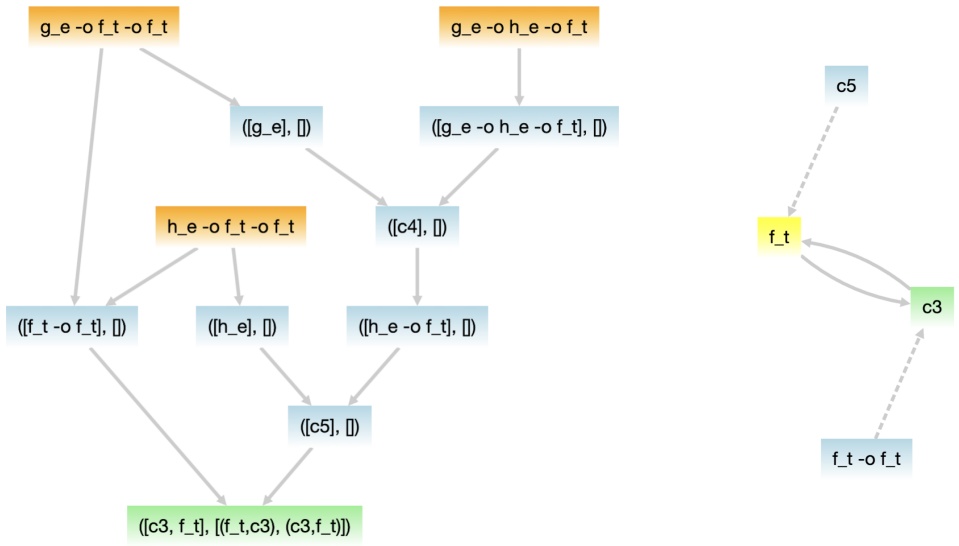
(30) Relevant categories:

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

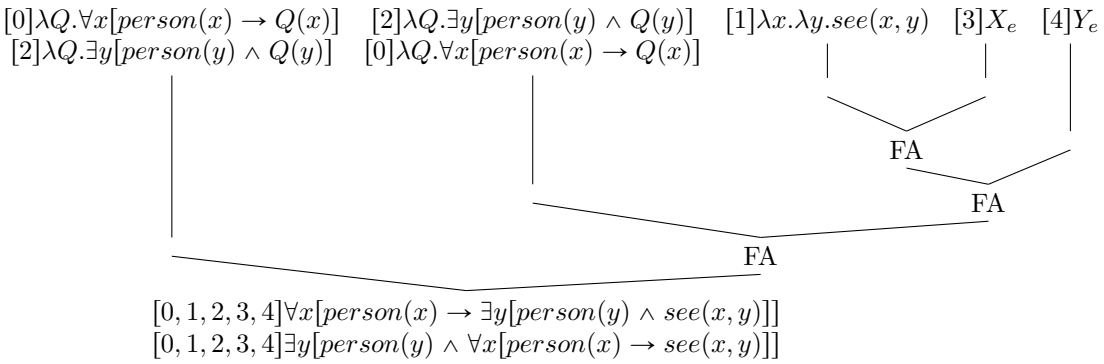
(31) a. History for category $g_e \multimap h_e \multimap f_t$:
$g_e \multimap h_e \multimap f_t \quad \rightarrow \quad \{\ h_1 : \lambda x.\lambda y.see(x,y) : g_e \multimap h_e \multimap f_t\ \}$

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

Linear logic derivation graph with strongly connected component in green



Resulting semantic derivation



A semantic derivation is a set of binary trees determining function applications steps (not syntactic trees, but cf. logical forms; Gotham 2018). During the linear logic derivation, combination histories are compressed so that common parts of the different solutions are only calculated once. Thus, graph-based glue proving factorizes out ambiguities (Maxwell III and Kaplan, 1993).

- Compositional ambiguity arise from SSCs (cycles)

  → SSCs describe the unordered elements in the partial order on function application steps

→ Ambiguity management should also happen in SCCs

> **Exercise 4**
>
> In this exercise you are tasked to explore the performance of the Hepple and the Lev prover. One potential drawback of the Lev prover is that it does not support QILL.

3.3 Constraining ambiguity in Lev (2007)

While the graph-based parser is more efficient than the chart-based parser by factoring out the ambiguities, it still generates exactly the same amount of solutions as the chart parser. I.e., it is still prone to spurious ambiguity. Thus, Lev (2007) proposes several strategies for constraining ambiguities.

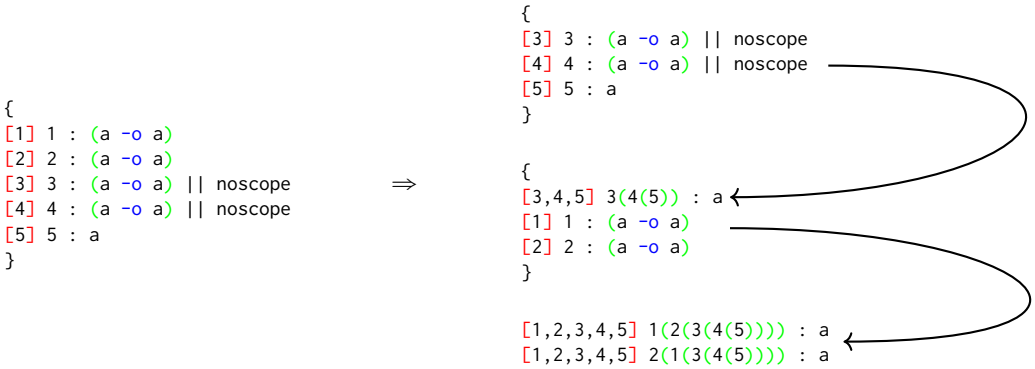
3.3.1 The `noscope` flag

The `noscope` flag uses two ingredients to constrain ambiguities. It is best used to avoid spurious ambiguities.

- **Partitioning** modifiers in SCCs into stacks of scoping and non-scoping modifiers
- apply modifiers in non-scoping stack in arbitrary order

  → **new proving routine**/alternatively filter results of chart prover
- add results to scoping stack and calculate non-scoping modifier as usual

An example:



3.3.2 Index-based ordering of modifiers

Index-based ordering of modifiers makes use of the the index sets used for tracking resource usage. Scope is restricted by ordering constraints of the form $x < y$, where x and y identify premise indices and x must outscope y, i.e. for (scope-taking) functions $f_{0\ldots n}$, $f_0(\ldots f_x(\ldots f_y(\ldots f_n(x))))$.

Agenda	**Chart**
$[1]1 : a \multimap a \mid\mid 1 < 2$	
$[2]2 : a \multimap a$	
$[3]3 : a$	

$\ldots$

Agenda	**Chart**
$[3]3 : a$	$[1]1 : a \multimap a \mid\mid 1 < 2$
	$[2]2 : a \multimap a$

Agenda	**Chart**
$[1,3]1(3) : a \mid\mid 1 < 2$	$[1]1 : a \multimap a \mid\mid 1 < 2$
$[2,3]2(3) : a$	$[2]2 : a \multimap a$
	$[3]3 : a$

Agenda	**Chart**
$[2,3]2(3) : a$	$[1]1 : a \multimap a \mid\mid 1 < 2$
	$[2]2 : a \multimap a$
	$[3]3 : a$
	$[1,3]1(3) : a \mid\mid 1 < 2$

Agenda	**Chart**
$[1,2,3]1(2(3)) : a$	$[1]1 : a \multimap a \mid\mid 1 < 2$
	$[2]2 : a \multimap a$
	$[3]3 : a$
	$[1,3]1(3) : a \mid\mid 1 < 2$
	$[2,3]2(3) : a$

Agenda	**Chart**
$\varnothing$	$[1]1 : a \multimap a \mid\mid 1 < 2$
	$[2]2 : a \multimap a$
	$[3]3 : a$
	$[1,3]1(3) : a \mid\mid 1 < 2$
	$[2,3]2(3) : a$
	$\mathbf{[1,2,3]1(2(3)) : a}$

- The chart algorithm keeps track of ordering constraints when combining functor and argument
- Constraints are propagated through derivation and removed when ascertained

→ Elevated role of index sets; important for the syntax/semantics interface

Exercise 5

In this exercise, we experiment with the `noscope` flag.

(32) **TASK 1:**
On the analysis page of the web interface, a very rudimentary event semantics is demonstrated. How does it make use of the `noscope` flag? To explore this, simply parse the default example and modify the resulting meaning constructors.

(33) **TASK 2:**
Parse the following sentences with the analysis tool. We already looked at examples like (33-b). Can we fix this using the `noscope` flag? Why? Why not?

a. A dog sniffed every tree.
b. A dog sniffed a tree

4 Day 3: The syntax/semantics interface

4.1 The syntax/semantics interface

		co-description	description-by-analysis
John	N	(↑PRED) = 'John' j :↑	`#f SUBJ #g PRED %g ==> #g GLUE %g : #g.`
Mary	N	(↑PRED) = 'Mary' m :↑	`#f OBJ #h PRED %h ==> #h GLUE %h : #h.`
loves	V	(↑PRED) = 'love<(↑SUBJ),(↑OBJ)>' $\lambda x.\lambda y.loves(x,y)$: (↑ SUBJ) ⊸ ((↑ OBJ) ⊸↑)	`#f SUBJ #g & #f OBJ #h & #f PRED %f` `==> #f GLUE %f : #g -o (#h -o #f).`

$$f\begin{bmatrix}\text{PRED} & \text{'love}\langle\text{SUBJ,OBJ}\rangle\text{'} \\ \text{SUBJ} & g\begin{bmatrix}\text{PRED} & \text{'John'}\end{bmatrix} \\ \text{OBJ} & h\begin{bmatrix}\text{PRED} & \text{'Mary'}\end{bmatrix}\end{bmatrix}$$

$j : g$
$m : h$
$\lambda x.\lambda y.loves(x,y) : g \multimap (h \multimap f)$

Figure 11: Co-descriptive lexicon vs. description-by-analysis rules

- **Co-description** is a term coined by researchers of LFG to describe an architecture where different aspects of language are described in parallel in a formally rigorous manner.
- **Description-by-analysis** is a procedural approach to deriving semantics from the syntax. The semantics are strongly linked to syntactic configurations.

5 Constraining ambiguity at the syntax/semantics interface

5.1 Multistage-proving (Findlay and Haug, 2022)

- Multistage proving proposes a new projection: the **proof structure**
- The proof structure is a **tree structure** organizing meaning constructors in a **cascading** fashion
- a node n identifies a proof tree; can be understood as landing site for (intermediate) solutions
- terminal nodes t, such that $n \prec t$, contain meaning constructors associated with n
- a node n may have one terminal node and arbitrarily many non-terminal nodes as daughters
- Proof structure nodes usually are derived via projections (mainly c- or f-structure)
- New nodes without c- and f-structure correspondent can be introduced, e.g., via local names

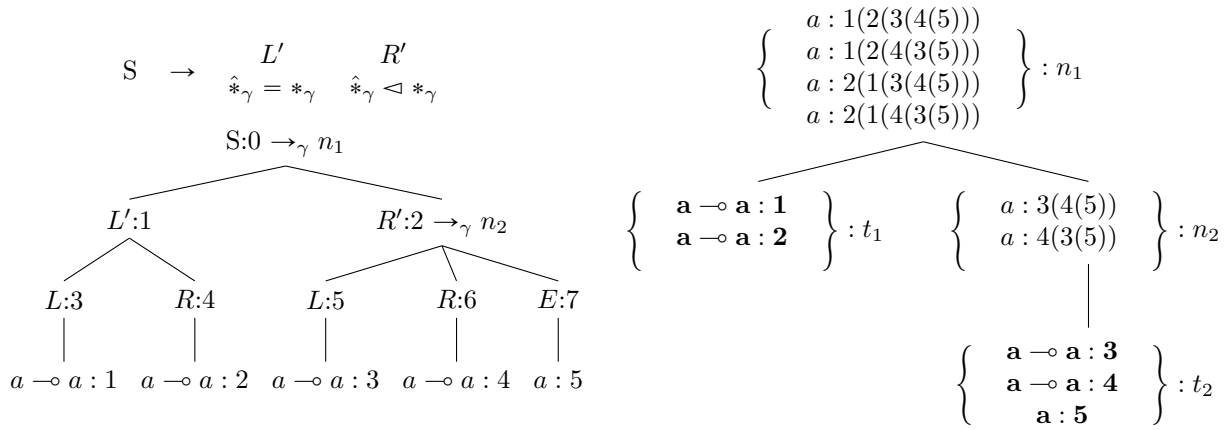


Figure 12: Relevant phrase structure rule, c-structure, and proof structure with added solutions

5.1.1 Theoretical status of the proof structure

Q: Is the proof structure a purely descriptive device or does it make any assumptions about how mcs can be grouped? (→ needs to be tested empirically.)

Possible constraints:

- Proof structures must be trees
  - A proof structure must form exactly one rooted tree for a given derivation (presumably Findlay and Haug's (2022) conception)
  - Constituents may be annotated with individual proof structures (leading to a disconnected proof structure)
- Dominance constraints must be congruent with c-structure (f-structure) dominance, i.e., is something like $*_\gamma \lhd \hat{*}_\gamma$ possible?
- If proof structures are allowed to project from f-structures, should we allow (inside-out) functional uncertainty dominance paths?

≈ Should it be impossible to enforce exceptional scope?

- The proof structure is translated into a bracketed set of meaning constructors for derivation

Bracketed set:

```
//4 solutions
{
1 : (a -o a)
2 : (a -o a)
{
3 : (a -o a)
4 : (a -o a)
5 : a
}
}
```

Regular set:

```
//24 solutions
{
1 : (a -o a)
2 : (a -o a)
3 : (a -o a)
4 : (a -o a)
5 : a
}
```

5.1.2 Multistage proving with the graph-based prover

- Due to factorizing out ambiguities, the graph-based prover can provide a derivation for a proof structure without any additional information (i.e., intermediate solutions)

  Procedure:

  - label all modifiers with an identifier corresponding to their position in the proof tree
  - When reaching a SCC, partition modifiers according to the proof structure labels
  - For each partition, calculate the solutions and feed them into the partion of the next higher label according to the proof structure (similar to the `noscope` flag)

- Modifiers are only removed from the derivation if they have been actually applied → they can escape proof structure boundaries if it fails otherwise

→ strict vs. relaxed proof structure?

(34) The following derivation succeeds in a relaxed proof structure:

a. $\exists x[dog(x) \wedge think(Kim, appear(x))]$

```
{
Kim : k_e
[/P_t.[/x_e.think(x,P)]] : (h_t -o (k_e -o f_t))
{
[/x_e.appear(x)] : (g_e -o h_t)
[/Q_<e,t>.Ex_e[dog(x) & Q(x)]] : ((g_e -o f_t) -o f_t)
}
}
```

- Partioning-based ambiguity management is fairly easy to implement as a cascade of chart derivations (as in Findlay and Haug 2022). The graph-based prover eliminates unwanted complexity.

References

Andrews, Avery D. 2008. The Role of PRED in LFG + Glue. In *Proceedings of the LFG08 Conference*, Pages 47–67.

Andrews, Avery D. 2018. Sets, Heads, and Spreading in LFG. *Journal of Language Modelling* 6.

Asudeh, Ash. 2004. *Resumption as Resource Management*. Ph.D. thesis, Stanford University.

Butt, Miriam, Tina Bögel, Mark-Matthias Zymla, and Benazir Mumtaz. 2024. Alternative questions in urdu: from the speech signal to semantics. In M. Butt, J. Y. Findlay, and I. Toivonen, editors., *Proceedings of the LFG'24 Conference*. Konstanz: PubliKon.

Chung, Sandra and William A Ladusaw. 2003. *Restriction and Saturation*. MIT press.

Cook, Philippa and John Payne. 2006. Information Structure and Scope in German. *LFG06* .

Crouch, Richard and Josef Van Genabith. 1999. Context Change, Underspecification, and the Structure of Glue Language Derivations. *Semantics and syntax in Lexical Functional Grammar: The resource logic approach* Pages 117–190.

Curry, Haskell Brooks, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. 1958. *Combinatory Logic*, volume 1. North-Holland Amsterdam.

Dalrymple, Mary, Agnieszka Patejuk, and Mark-Matthias Zymla. 2020. XLE+Glue – A New Tool for Integrating Semantic Analysis in XLE. In M. Butt, T. H. King, and I. Toivonen, editors., *Proceedings of the LFG'20 Conference, Australian National University*. Stanford, CA: CSLI Publications.

de Groote, Philippe. 1999. An Algebraic Correctness Criterion for Intuitionistic Multiplicative Proof-nets. *Theoretical Computer Science* 224(1-2):115–134.

Dipper, Stefanie. 2003. *Implementing and Documenting Large-scale Grammars-German LFG*. Ph.D. thesis, University of Stuttgart.

Falk, Yehuda. 2011. *Lexical-Functional Grammar*. Oxford University Press.

Findlay, Jamie and Dag Haug. 2022. Managing Scope Ambiguities in Glue via Multistage Proving. In *Proceedings of the Lexical Functional Grammar Conference*, Pages 144–163.

Gotham, Matthew. 2018. Making Logical Form Type-logical: Glue semantics for Minimalist syntax. *Linguistics and Philosophy* 41(5):511–556.

Gotham, Matthew. 2020. Constraining Scope Ambiguity in LFG + Glue .

Gotham, Matthew. 2022. Approaches to Scope Islands in LFG+Glue .

Gupta, Vineet and John Lamping. 1998. Efficient linear logic meaning assembly. In *Proceedings of the 17th international conference on Computational linguistics-Volume 1*, Pages 464–470. Association for Computational Linguistics.

Hepple, Mark. 1996. A compilation-chart method for linear categorial deduction. In *Proceedings of the 16th conference on Computational linguistics-Volume 1*, Pages 537–542. Association for Computational Linguistics.

Kaplan, Ronald M. 1995. Three Seductions of Computational Psycholinguistics. *Formal Issues in Lexical-Functional Grammar* 47.

Kaplan, Ronald M and Jürgen Wedekind. 2023. Formal and Computational Properties of LFG. *Handbook of Lexical Functional Grammar* Pages 1035–1082.

Kratzer, Angelika and Irene Heim. 1998. *Semantics in Generative Grammar*. Blackwell Oxford.

Kuhn, Jonas. 1998. *Resource Sensitivity in the Syntax-semantics Interface and the German split NP Construction*. Universität Stuttgart, Fakultät Philosophie.

Lev, Iddo. 2007. *Packed Computation of Exact Meaning Representations*. Ph.D. thesis, Stanford University.

Maxwell III, John T and Ronald M Kaplan. 1993. The Interface between Phrasal and Functional Constraints. *Computational Linguistics* 19(4):571–590.

Meßmer, Moritz and Mark-Matthias Zymla. 2018. The Glue Semantics Workbench: A Modular Toolkit for Exploring Linear Logic and Glue Semantics. In M. Butt and T. H. King, editors., *Proceedings of the LFG'18 Conference, University of Vienna*, Pages 249–263. Stanford, CA: CSLI Publications.

Moot, Richard and Christian Retoré. 2012. *The Logic of Categorial Grammars: a Deductive Account of Natural Language Syntax and Semantics*, volume 6850. Springer.

Perrier, Guy. 1999. Labelled Proof Nets for the Syntax and Semantics of Natural Languages. *Logic Journal of the IGPL* 7(5):629–654.

Saraswat, Vijay. 1999. LFG as Concurrent Constraint Programming. *Semantics and Syntax in Lexical Functional Grammar: The Resource Logic Approach* Pages 281–318.

Zymla, Mark-Matthias. 2024a. Ambiguity management in computational Glue semantics. In M. Butt, J. Y. Findlay, and I. Toivonen, editors., *Proceedings of the LFG'24 Conference*, Pages 285–310. Konstanz, Germany: PubliKon.

Zymla, Mark-Matthias. 2024b. *Tense and Aspect in Multilingual Semantic Construction*. Ph.d. dissertation, University of Konstanz, Konstanz, Germany.

A Computational tools for Glue semantics

This handout is based on the resources developed in Dalrymple et al. (2020); Meßmer and Zymla (2018); Zymla (2024b), developing into a Glue semantics library.

- Glue Semantics Workbench (GSWB): a sample of glue semantics provers written in Java
- Linguistic Graph Expansion and Rewriting (LiGER): description-by-analysis and other utilities for XLE+Glue
- XLE+Glue: Semantic capabilities for XLE

A.1 Other implementations

- Pretty Print Web Prover[1]: developed by Gianluca Giorgolo linear implication and multiplicative conjunction; available for testing online
- Instant Glue Prover[1][2]: developed by Miltiadis Kokkonidis; linear implication and multiplicative conjunction; only 70 lines of code
- Baby Glue[3]: developed by Avery Andrews; implicational fragment; based on work in de Groote (1999); Perrier (1999)

[1] Provided via Ash Asudeh at `http://www.sas.rochester.edu/lin/sites/asudeh/research/prover.html`.
[2] See also: `https://github.com/saraswat/instant-glue`
[3] Available at `http://averyandrews.net/Software/`.

B Computational complexity

The worst case

```
x : AX_t.(X_t -o X_t)
x : AY_t.(Y_t -o Y_t)
x : AZ_t.(Z_t -o Z_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
```

```
{
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
```

```
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
```

```
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
{
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
a : (a_t -o a_t)
b : (b_t -o b_t)
b : (b_t -o b_t)
c : (a_t -o b_t)
d : a_t
}
```

Nested quantifiers

QILL:

```
{
every : AX_t.((g_e -o g_t) -o ((g_e -o X_t) -o X_t))
rep : (g_e -o g_t)
of : ((g_e -o g_t) -o (g_e -o (j_e -o g_t)))
a-company : AY_t.((j_e -o Y_t) -o Y_t)
a-sample : AZ_t.((h_e -o Z_t) -o Z_t)
see : (g_e -o (h_e -o f_t))
}
```

ILL:

```
{
every : ((g_e -o g_t) -o ((g_e -o f_t) -o f_t))
rep : (g_e -o g_t)
of : ((g_e -o g_t) -o (g_e -o (j_e -o g_t)))
a-company : ((j_e -o g_t) -o g_t)
a-sample : ((h_e -o f_t) -o f_t)
see : (g_e -o (h_e -o f_t))
}
```

```
{
every : ((g_e -o g_t) -o ((g_e -o f_t) -o f_t))
rep : (g_e -o g_t)
of : ((g_e -o g_t) -o (g_e -o (j_e -o g_t)))
a-company : ((j_e -o f_t) -o f_t)
a-sample : ((h_e -o f_t) -o f_t)
see : (g_e -o (h_e -o f_t))
}
```

C Multi-stage proving

Simplified pseudo code for extracting proof structures

```
# proof_constraints are equals or dominance constraints as encoded in the input
# proof_constraint = {proof_constraint.node, proof_constraint.elements,
                      proof_constraint.daughter.node, proof_constraint.daughter.elements}
proof_constraints = []

# proof_tree is the goal structure: a map whose entries form a tree and whose terminal nodes are sets of mcs
proof_tree = {}

"""
Params:
cstr: at most binary branching tree -> {cstr.root, cstr.left, cstr.right}
active_node: a string
"""
function traverseCStructure(cstr, active_node):
    current_node = cstr.root
    proof_constraint = find_proof_constraint(current_node)

    if proof_constraint:
        proof_constraints.addFirst(proof_constraint)
        active_node = proof_constraint.node
    else:
        # Determine active_node
        for pc in proof_constraints:
            if current_node in pc.elements:
                active_node = pc.node
                break
            else:
                for daughter in pc.daughters:
                    if current_node in daughter.elements:
                        if active_node not in proof_tree:
                            proof_tree[active_node] = []
                        proof_tree[active_node].add(daughter.node)
                        active_node = daughter.node
                        break
                if active_node == daughter.node:
                    break

        # Associate meaning constructors with active node
        mcs = find_mcs(current_node)
        if active_node not in proof_tree:
            proof_tree[active_node] = []
        proof_tree[active_node].addAll(mcs)

    if cstr.isBranching():
        # Traverse right daughter first
        traverseCStructure(cstr.right, active_node)
    # Traverse left/single daughter
    traverseCStructure(cstr.left, active_node)
```

D Additional proofs in GSWB format

```
Jordan : g_e
[/x_e.[/y_e.like(x,y)]] : (g_e -o (h_e -o f_t))
Alex : h_e

[/Q_<e,t>.Ex_e[person(x) -> Q(x)]] : ((g_e -o f_t) -o f_t)
[/x_e.[/y_e.like(x,y)]] : (g_e -o (h_e -o f_t))
[/Q_<e,t>.Ex_e[person(x) & Q(x)]] : ((h_e -o f_t) -o f_t)

{
Kim : k_e
[/P_t.[/x_e.think(x,P)]] : (h_t -o (k_e -o f_t))
[/x_e.appear(x)] : (g_e -o h_t)
[/Q_<e,t>.Ex_e[dog(x) & Q(x)]] : ((g_e -o h_t) -o h_t)
}
{
Kim : k_e
[/P_t.[/x_e.think(x,P)]] : (h_t -o (k_e -o f_t))
[/x_e.appear(x)] : (g_e -o h_t)
[/Q_<e,t>.Ex_e[dog(x) & Q(x)]] : ((g_e -o f_t) -o f_t)
}
{
Kim : k_e
[/P_t.[/x_e.think(x,P)]] : (h_t -o (k_e -o f_t))
{
[/x_e.appear(x)] : (g_e -o h_t)
[/Q_<e,t>.Ex_e[dog(x) & Q(x)]] : ((g_e -o f_t) -o f_t)
}
}

//4 solutions
{
1 : (a -o a)
2 : (a -o a)
{
3 : (a -o a)
4 : (a -o a)
5 : a
}
}
//24 solutions
{
1 : (a -o a)
2 : (a -o a)
3 : (a -o a)
4 : (a -o a)
5 : a
}

// plain proof
{
mary : m_e
[/Q_<e,t>.Ex_e[student(x) & Q(x)]] : ((s_e -o f_t) -o f_t)
[/Q_<e,t>.Ax_e[grade(x) -> Q(x)]] : ((g_e -o f_t) -o f_t)
[/x_e.[/y_e.[/z_e.give(x,z,y)]]] : (m_e -o (s_e -o (g_e -o f_t)))
}
// multistage proving
{
mary : m_e
[/Q_<e,t>.Ex_e[student(x) & Q(x)]] : ((s_e -o f_t) -o f_t)
{
[/Q_<e,t>.Ax_e[grade(x) -> Q(x)]] : ((g_e -o f_t) -o f_t)
[/x_e.[/y_e.[/z_e.give(x,z,y)]]] : (m_e -o (s_e -o (g_e -o f_t)))
}
}
```

D.1 Interleaved semantic parsing

- Interleaved parsing was described, e.g., in Maxwell III and Kaplan (1993) for c- and f-structures
- It describes the idea of pruning solutions early by calculating f-descriptions and c-descriptions in parallel

≈ bottom up parsing of individual constituents and rule out those constituents that fail early (relies on monotonicity)

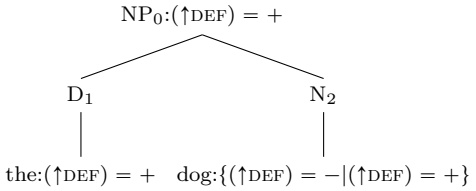
> **Monotonicity:**
> A system of constraints is *monotonic* if no deduction is ever retracted when new constraints are conjoined.
>
> (Maxwell III and Kaplan, 1993)

- Existential constraints and constraining equations are not strictly monotonic (e.g., Falk 2011; Saraswat 1999)

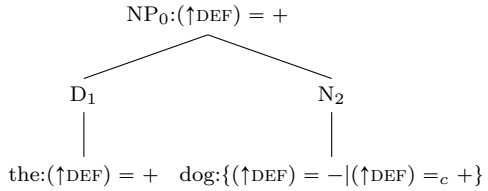
$$\text{NP} \rightarrow \begin{matrix}\text{D}\\(\uparrow\text{DEF}) = +\end{matrix} \quad \begin{matrix}\text{NP}\\\{(\uparrow\text{DEF}) = -|\\(\uparrow\text{DEF}) = +\}\end{matrix}$$

$$\text{NP} \rightarrow \begin{matrix}\text{D}\\(\uparrow\text{DEF}) = +\end{matrix} \quad \begin{matrix}\text{NP}\\\{(\uparrow\text{DEF}) = -|\\(\uparrow\text{DEF}) =_c +\}\end{matrix}$$

(35)



(36)

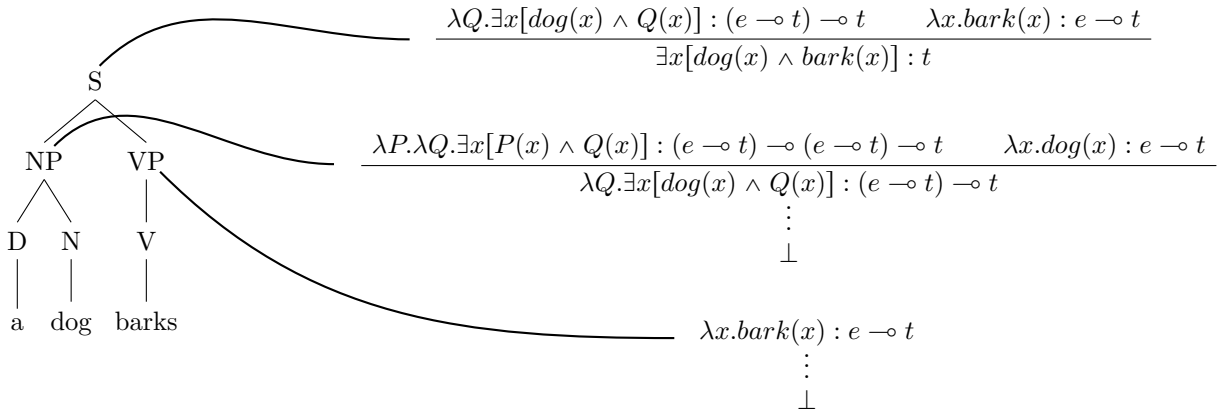


- In (35), at N_2, there are two plausible solutions but one deduction fails in the NP
- In (36), at N_2, only one deduction is possible
  - the second disjunct fails because DEF: is not defined
    → adding D_1 in (36) recovers the failed deduction
  - the other deduction fails as in (35)

⇒ **Non-monotonicity enables us to recover "failed" analyses**

(37) f-structure felicity conditions are stated non-monotonically
  a. Completeness: (↑ SUBJ) (↑ OBJ)
  b. Coherence: ¬(↑ COMP) ¬(↑ OBL) ¬(↑ XCOMP) ... (Kaplan and Wedekind, 2023)

(38) a.



  b. (↑ GF) ≈ (↑ GF) ⊸↑ (cf., e.g., Andrews 2008; Asudeh 2004; Kuhn 1998)

⇒ **Glue semantics is non-monotonic. This undermines interleaved parsing (but doesn't make it necessarily impossible).**

D.2 Comparison to theoretic proposals to constraining scope (Gotham 2020 and Gotham 2022)

While ideas on constraining ambiguities pop up in various works (e.g., Andrews 2018; Cook and Payne 2006; Crouch and Van Genabith 1999), this section focuses on the research conducted by Gotham (2020, 2022) which can be broadly classified as index/parameter-based approaches to constraining ambiguity.

- Gotham's proposals extend the linear logic fragment in one way or another

→ Ambiguity management is semantically autonomous

D.2.1 A brief discussion of Gotham (2020)

- Glue resources are indexed to further constrain combinatory possibilities

  ≈ only resources with matching indices can be consumed

- Indices are essentially instantiated relative to the result of the derivation (f_0 in the examples below)

(39) Ein Polizist bewacht jeden Ausgang.
every officer guard an exit

a. $\exists x[\mathit{officer}(x) \wedge \forall y[\mathit{exit}(y) \rightarrow \mathit{guard}(y, x)]]$
b. ~~$\forall y[\mathit{exit}(y) \rightarrow \exists x[\mathit{officer}(x) \wedge \mathit{guard}(x, y)]]$~~

(40) **Successful proof for example (39-a)**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[x : g_1]^1}\quad \dfrac{\dfrac{\lambda x.\lambda y.\mathit{guard}(x,y) :}{\forall l.\forall \eta.g_l \multimap (h_\eta \multimap f_\eta)}}{\lambda x.\lambda y.\mathit{guard}(x,y) :}\ \forall_E}{g_1 \multimap (h_2 \multimap f_2)}}{\lambda y.\mathit{guard}(X,y) : h_2 \multimap f_2}\quad \dfrac{\text{jeden Ausgang}}{\lambda Q.\forall y[\mathit{exit}(y) \rightarrow Q(y)]}}{(h_2 \multimap f_2) \multimap f_1}}{\forall y[\mathit{exit}(y) \rightarrow \mathit{guard}(X,y)] : f_1}}{\lambda x.\forall y[\mathit{exit}(y) \rightarrow \mathit{guard}(x,y)] : g_1 \multimap f_1}\quad \dfrac{\text{Ein Polizist}}{\lambda Q.\exists x[\mathit{officer}(x) \wedge Q(x)] : (g_1 \multimap f_1) \multimap f_0}$$

$$\exists x[\mathit{officer}(x) \wedge \forall y[\mathit{exit}(y) \rightarrow \mathit{guard}(x,y)]] : f_0$$

(41) **Failed proof for example (39-b)**

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{[Y : h_1]^2}\quad \dfrac{\dfrac{}{[X : g_2]^1}\quad \dfrac{\dfrac{\lambda x.\lambda y.\mathit{guard}(x,y) :}{\forall l.\forall \eta.g_l \multimap (h_\eta \multimap f_\eta)}}{\lambda x.\lambda y.\mathit{guard}(x,y) :}\ \forall_E}{g_2 \multimap (h_1 \multimap f_1)}}{\lambda y.\mathit{guard}(X,y) : h_1 \multimap f_1}}{\mathit{guard}(X,Y) : f_1}}{\lambda x.\mathit{guard}(x,Y) : g_2 \multimap f_1}\quad \dfrac{\text{Ein Polizist}}{\lambda Q.\exists x[\mathit{officer}(x) \wedge Q(x)] : (g_2 \multimap f_2) \multimap f_1}\ *$$

Caveat 1:

- Keeping track of indices is complicated as assumptions are co-indexed with the quantifiers that discharge them but nodes are not generally co-indexed (also see quantifiers)

(42) Jeden Ausgang bewacht ein Polizist.
An exit guard every officer

a. $\exists x[\mathit{officer}(x) \wedge \forall y[\mathit{exit}(y) \rightarrow \mathit{guard}(y, x)]]$
b. $\forall y[\mathit{exit}(y) \rightarrow \exists x[\mathit{officer}(x) \wedge \mathit{guard}(x, y)]]$

- Example (42) allows for both readings, (40) and (41). To account for this, Gotham implements the RESET template as a repair mechanism (fixed version of (41) for reading (42-b) in appendix 3.1.2)

(43) RESET := $(\uparrow \text{OBJ}) <_f (\uparrow \text{SUBJ})$
$\lambda p.p : \forall l.\forall \eta.\ \uparrow_l \multimap \uparrow_\eta$

Caveat 2:

- Ultimately, f-structure conditions are used to allow for certain readings; adds additional modifiers

Bottom line:

→ Autonomous semantics but high information density and computationally complex indexation mechanism; yet still not completely independent of syntax[4]

[4] Gotham (2022) is arguably more elegant but still requires (multiple) indexations to keep track off. Thus, the information density is still high.

Gotham (2020) example derivation with reset template

$$\dfrac{\dfrac{\dfrac{\dfrac{[Y : h_1]^2,\ [X : g_2]^1 \qquad \begin{array}{c}\text{bewacht}\\ \vdots \\ \lambda x.\lambda y.guard(x,y) : \\ g_2 \multimap h_1 \multimap f_1\end{array}}{guard(X,Y) : f_1} \qquad \dfrac{\begin{array}{c}\text{(RESET)}\\ \lambda p.p : \\ \forall l.\forall \eta.f_l \multimap f_\eta\end{array}}{\begin{array}{c}\lambda p.p : \\ f_1 \multimap f_2\end{array}}\forall_E}{guard(X,Y) : f_2}}{\lambda x.guard(x,Y) : g_2 \multimap f_2} \qquad \begin{array}{c}\text{Ein Polizist}\\ \vdots \\ \lambda Q.\exists x[officer(x) \wedge Q(x)] : \\ (g_2 \multimap f_2) \multimap f_1\end{array}}{\dfrac{\exists x[officer(x) \wedge guard(x,Y)] : f_1}{\lambda y.\exists x[officer(x) \wedge guard(x,y)] : h_1 \multimap f_1}} \qquad \begin{array}{c}\text{jeden Ausgang}\\ \vdots \\ \lambda Q.\forall y[exit(y) \rightarrow Q(y)] : \\ (h_1 \multimap f_1) \multimap f_0\end{array}}{\forall y[exit(y) \rightarrow \exists x[officer(x) \wedge guard(x,y)]] : f_0}$$

E Lessons learned

E.1 Formal properties of ambiguity in Glue semantics

Q: Does order matter at the level of semantic derivation?

Q: What properties does the logic underlying natural language semantic composition have?

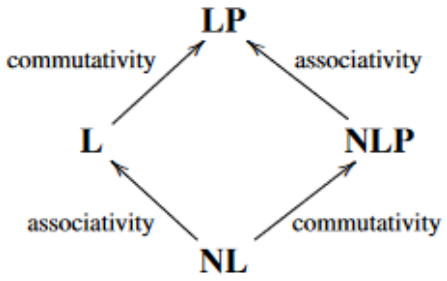


Figure 13: Kinds of type logics (Gotham, 2022)[5]

- LP ≈ ILL = associative and commutative

(44) a. **associative:** meaning constructors can be freely regrouped

$$\frac{(\Gamma, \Delta), \Sigma \vdash A}{\Gamma, (\Delta, \Sigma) \vdash A}\ \text{ASSOC}$$

b. **commutative:** meaning constructors can be freely reordered

$$\frac{\Gamma, \Delta \vdash A}{\Delta, \Gamma \vdash A}\ \text{COMM}$$

- Controlled non-associativity seems to be the property to explore
  - Proof trees (Findlay and Haug, 2022) and Gotham (2022) propose to make Glue semantics partially non-associative[6]
  - Gotham (2022) discusses strategies for dealing with scope freezing and scope blocking proposing a modification of the underlying linear logic fragment making it effectively multi-modal
  - Asudeh (2004, p. 81) criticizes multi-modal approaches:[7]

(45) "*This is certainly an option, but faces the danger of* ***conflating properties of syntactic and semantic combination*** *by failing to separate syntax, where order is fairly relevant, from semantics, where order is irrelevant. There may be* ***complexities*** *that arise in controlling syntactic or semantic combination, but these* ***will not be localized in syntax or semantics*** *and will* ***instead infect the system as a whole****.*"

- commutativity is sensitive to the meaning side of a meaning constructor, i.e., linear logic is commutative but meaning constructors aren't necessarily

Q: How are semantics made non-associative?

E.2 Spilling over

(46) **Hypotheses:**

a. Non-monotonic aspects of f-structure such as defining and constraining equations are spilling into f-structure from semantics

b. Structural aspects of syntax spill over into semantics inducing non-associativity

[5] Based on Moot and Retoré (2012)

[6] cf. also Moot and Retoré (2012, p.111ff): "*What we would like is to have some sort of controlled access to the structural rules of associativity and commutativity.*"

[7] Highlights are my own

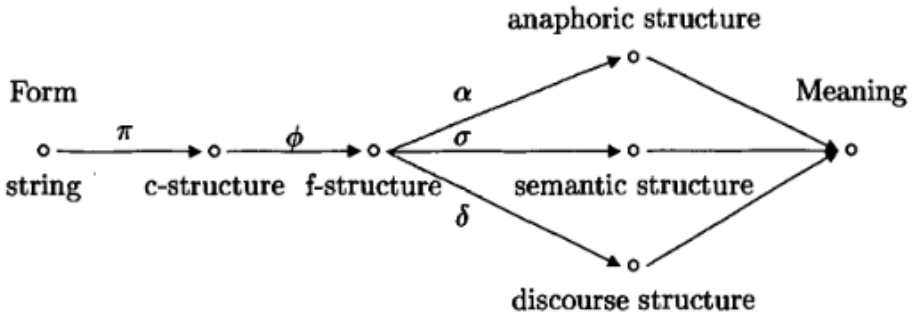


Figure 14: Grammar in Kaplan (1995)

- Computational complexity increases from left-to-right in a linear form to meaning mapping (e.g., Figure 14)
- Sometimes calculations can be **pushed to the left** to reduce computational complexity

C-structure	**F-structure**	**Glue derivation**
polynominal	exponential	factorial

Table 3: Increasing complexity in the form-to-meaning mapping

Two examples:

- **Constraining equations** mimic apparent resource sensitivity at f-structure but also introduce non-monotonic aspects
- **Parameterized rules** use complex categories to shift feature unification from f-structure to c-structure (Dipper, 2003)

The other direction

Q: Is order spilling over from the syntax into the semantics?

- Approaches like the proof structure suggest *yes*!

- Gotham (2022, p. 17) points out that semantic properties of quantifiers, e.g., monotonicity constrain certain scope freezing effects
  - Ultimately, Gotham does not explicitly implement monotonicity constraints but rather constraints on argument order
  - somewhat relatedly, Lev (2007) proposes to constrain ambiguity in the semantics by determining quantifier strength ignoring ambiguities between quantifiers of the same strength (e.g. $\exists x[\exists y[see(x, y)]]$)

→ Ultimately, spilling over distributes complexity across the form-to-meaning mapping, improving computability. Spilling over is bi-directional

→ Autonomy of syntax and semantics are formally elegant but computational approaches at least cast doubt on their viability beyond descriptive clarity

→ While not explored in depth in this paper, parameterized rules suggest that spilling over may also allow for more formally elegant analyses of certain linguistic facts

F LFG

LFG is well known for distributing information across multiple representations. The idea behind this is that different parts of language follow different rules and, thus, deserve descriptions in a formalism that suits these rules. This distinguishes it from linguistics following the Chomskian tradition in the sense that there is less (obvious) reliance on tree structures. It also distinguishes LFG from incremental models, e.g., CCG, which is driven by type-logical considerations but conflates syntax and semantics (to a larger degree than LFG).

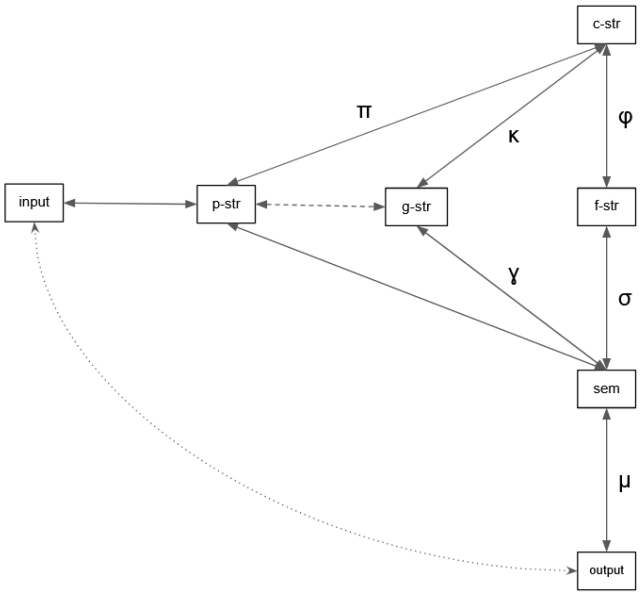


Figure 15: Computational projection architecture

- XLE+Glue (Dalrymple et al., 2020)
- g-structure (discussed as proof structure; Findlay and Haug, 2022; Zymla, 2024a)
- p-structure/s-structure correspondence (Butt et al., 2024)

Conclusion: Conceptually, constraining ambiguity seems like a simple task. However, it becomes increasingly complex as more aspects of linguistic knowledge are represented in formal representations. Generally, it seems like the knowledge is best evenly distributed across various projections which does not only enhance explainability and descriptive clarity but also seems to make more sense computationally as it avoids spikes in complexity for any one level of representation. Thus, projections should not be understood as autonomous levels of a linguistic discipline (e.g., syntax/semantics). This conclusion has been drawn on the basis of exploring the syntax/semantics interface from a theoretical and mainly computational perspective.