Approaches to scope islands in LFG+Glue

Matthew Gotham
University of Oxford

26th International LFG Conference
13–16 July 2021

Outline

Background

Outline

Background

Scope islands

Outline

Outline

Background

Scope islands

Approaching the data

- Blocking features and off-path constraints
- Multi-modal Glue semantics

Discussion

Background

(1) Jim smiles.

$$f : \begin{bmatrix} \textsc{pred} & \text{`smile'} \\ \textsc{subj} & g : [\ \text{"Jim"}\ ] \end{bmatrix}$$

$$\textit{Jim} \rightsquigarrow \mathbf{jim} : {\uparrow}_e$$

$$\textit{smiles} \rightsquigarrow \mathbf{smile} : (\uparrow \textsc{subj})_e \multimap {\uparrow}_p$$

(1) Jim smiles.

$$f : \begin{bmatrix} \textsc{pred} & \text{`smile'} \\ \textsc{subj} & g : [\text{ ``Jim'' }] \end{bmatrix}$$

$$\textit{Jim} \rightsquigarrow \mathbf{jim} : g_e$$

$$\textit{smiles} \rightsquigarrow \mathbf{smile} : g_e \multimap f_p$$

(1) Jim smiles.

$$f : \begin{bmatrix} \text{PRED} & \text{`smile'} \\ \text{SUBJ} & g : [\text{ ``Jim'' }] \end{bmatrix}$$

$$\textit{Jim} \rightsquigarrow \mathbf{jim} : g_e$$
$$\textit{smiles} \rightsquigarrow \mathbf{smile} : g_e \multimap f_p$$

$$\frac{\mathbf{jim} : g_e \qquad \mathbf{smile} : g_e \multimap f_p}{\mathbf{smile}(\mathbf{jim}) : f_p} \multimap_E$$

Scope ambiguity

(2) Someone sees everything.

$\Rightarrow$ $\mathsf{someone}(\lambda x.\mathsf{everything}(\lambda y.\mathsf{see}(x, y)))$ (surface scope)

$\Rightarrow$ $\mathsf{everything}(\lambda y.\mathsf{someone}(\lambda x.\mathsf{see}(x, y)))$ (inverse scope)

Scope ambiguity

(2) Someone sees everything.

$\Rightarrow$ $\textbf{someone}(\lambda x.\textbf{everything}(\lambda y.\textbf{see}(x, y)))$ (surface scope)

$\Rightarrow$ $\textbf{everything}(\lambda y.\textbf{someone}(\lambda x.\textbf{see}(x, y)))$ (inverse scope)

$$f : \begin{bmatrix} \textsc{pred} & \text{‘see}\langle g, h\rangle\text{’} \\ \textsc{tense} & \textsc{pres} \\ \textsc{subj} & g : \begin{bmatrix} \textsc{pred} & \text{‘someone’} \end{bmatrix} \\ \textsc{obj} & h : \begin{bmatrix} \textsc{pred} & \text{‘everything’} \end{bmatrix} \end{bmatrix}$$

Multiple proofs

$$f : \begin{bmatrix} \textsc{pred} & \text{`see}\langle g, h\rangle\text{'} \\ \textsc{tense} & \textsc{pres} \\ \textsc{subj} & g : \begin{bmatrix} \textsc{pred} & \text{`someone'} \end{bmatrix} \\ \textsc{obj} & h : \begin{bmatrix} \textsc{pred} & \text{`everything'} \end{bmatrix} \end{bmatrix}$$

$$\begin{aligned} \textit{someone} &\rightsquigarrow \textbf{someone} : (\uparrow_e \multimap (\textsc{gf}\uparrow)_p) \multimap (\textsc{gf}\uparrow)_p \\ \textit{sees} &\rightsquigarrow \lambda y.\lambda x.\textbf{see}(x, y) : (\uparrow \textsc{obj})_e \multimap ((\uparrow \textsc{subj})_e \multimap \uparrow_p) \\ \textit{everything} &\rightsquigarrow \textbf{everything} : (\uparrow_e \multimap (\textsc{gf}\uparrow)_p) \multimap (\textsc{gf}\uparrow)_p \end{aligned}$$

Multiple proofs

$$f : \begin{bmatrix} \textsc{pred} & \text{`see}\langle g, h\rangle\text{'} \\ \textsc{tense} & \textsc{pres} \\ \textsc{subj} & g : \begin{bmatrix} \textsc{pred} & \text{`someone'} \end{bmatrix} \\ \textsc{obj} & h : \begin{bmatrix} \textsc{pred} & \text{`everything'} \end{bmatrix} \end{bmatrix}$$

$$\begin{aligned} \textit{someone} &\rightsquigarrow \textbf{someone} : (g_e \multimap f_p) \multimap f_p \\ \textit{sees} &\rightsquigarrow \lambda y.\lambda x.\textbf{see}(x, y) : h_e \multimap (g_e \multimap f_p) \\ \textit{everything} &\rightsquigarrow \textbf{everything} : (h_e \multimap f_p) \multimap f_p \end{aligned}$$

$$
\dfrac{\begin{array}{c}\text{someone} :\\ (g_e \multimap f_p) \multimap f_p\end{array} \qquad \dfrac{\dfrac{\begin{array}{c}\text{everything} :\\ (h_e \multimap f_p) \multimap f_p\end{array} \qquad \dfrac{\dfrac{\dfrac{\begin{array}{c}\lambda v.\lambda u.\text{see}(u,v) :\\ h_e \multimap (g_e \multimap f_p)\end{array} \quad [y : h_e]^1}{\lambda u.\text{see}(u,y) : g_e \multimap f_p} \qquad [x : g_e]^2}{\text{see}(x,y) : f_p}}{\lambda y.\text{see}(x,y) : h_e \multimap f_p}\,1}{\text{everything}(\lambda y.\text{see}(x,y)) : f_p}}{\lambda x.\text{everything}(\lambda y.\text{see}(x,y)) : g_e \multimap f_p}\,2}{\text{someone}(\lambda x.\text{everything}(\lambda y.\text{see}(x,y))) : f_p}
$$

Inverse scope interpretation

$$
\dfrac{
\begin{array}{c}\text{everything} :\\ (h_e \multimap f_p) \multimap f_p\end{array}
\qquad
\dfrac{
\dfrac{
\begin{array}{c}\text{someone} :\\ (g_e \multimap f_p) \multimap f_p\end{array}
\qquad
\dfrac{
\begin{array}{c}\lambda v.\lambda x.\text{see}(x, v) :\\ h_e \multimap (g_e \multimap f_p)\end{array}
\quad
[y : h_e]^1
}{
\lambda x.\text{see}(x, y) : g_e \multimap f_p
}
}{
\text{someone}(\lambda x.\text{see}(x, y)) : f_p
}
}{
\lambda y.\text{someone}(\lambda x.\text{see}(x, y)) : h_e \multimap f_p
}\;1
}{
\text{everything}(\lambda y.\text{someone}(\lambda x.\text{see}(x, y))) : f_p
}
$$

Other manifestations of scope ambiguity

Embedded quantified noun phrases:

(3) A member of every board resigned.

Other manifestations of scope ambiguity

Embedded quantified noun phrases:

(3) A member of every board resigned.

$\Rightarrow$ $\mathsf{some}(\lambda x.\mathsf{every}(\mathsf{board}, \lambda y.\mathsf{member\text{-}of}(x, y)), \mathsf{resign})$

surface scope

Other manifestations of scope ambiguity

Embedded quantified noun phrases:

(3) A member of every board resigned.

$\Rightarrow$ $\mathsf{some}(\lambda x.\mathsf{every}(\mathsf{board}, \lambda y.\mathsf{member\text{-}of}(x, y)), \mathsf{resign})$

surface scope

$\Rightarrow$ $\mathsf{every}(\mathsf{board}, \lambda y.\mathsf{some}(\lambda x.\mathsf{member\text{-}of}(x, y), \mathsf{resign}))$

inverse linking

$$f : \begin{bmatrix} \text{PRED} & \text{`resign}\langle g \rangle\text{'} \\ \text{SUBJ} & g : \begin{bmatrix} \text{PRED} & \text{`member}\langle h \rangle\text{'} \\ \text{SPEC} & \begin{bmatrix} \text{PRED} & \text{`a'} \end{bmatrix} \\ \text{OBJ} & h : \begin{bmatrix} \text{“every board”} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\textit{every board} \rightsquigarrow \lambda P.\mathbf{every}(\mathbf{board}, P) : (\uparrow_e \multimap ?_p) \multimap ?_p$$

Surface scope : $? := g$

Inverse linking : $? := f$

How to fix scope level?

$$f : \begin{bmatrix} \ldots & \\ \textsc{subj} & g : \begin{bmatrix} \ldots & \\ \textsc{obj} & h : \begin{bmatrix} \text{“every board”} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Two methods:

1. Inside-out functional uncertainty:

   $\%A = (\textsc{path} \uparrow)$

   $\lambda P.\mathbf{every}(\mathbf{board}, P) : (\uparrow_e \multimap \%A_p) \multimap \%A_p$

How to fix scope level?

$$f : \begin{bmatrix} \ldots & \\ & \\ \textsc{subj} & g : \begin{bmatrix} \ldots & \\ \textsc{obj} & h : \begin{bmatrix} \text{“every board”} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

Two methods:

1. Inside-out functional uncertainty:

   $\%A = (\textsc{path} \uparrow)$

   $\lambda P.\mathbf{every}(\mathbf{board}, P) : (\uparrow_e \multimap \%A_p) \multimap \%A_p$

2. Quantification in linear logic:

   $\lambda P.\mathbf{every}(\mathbf{board}, P) : \forall X.(\uparrow_e \multimap X_p) \multimap X_p$

Scope islands

Limitations on scope level

(4) A warden thinks that every prisoner escaped.

$\Rightarrow$ $\mathrm{some}(\mathrm{warden}, \lambda x.\mathrm{think}(x, \mathrm{every}(\mathrm{prisoner}, \mathrm{escape})))$

$\not\Rightarrow$ $\mathrm{every}(\mathrm{prisoner}, \lambda y.\mathrm{some}(\mathrm{warden}, \lambda x.\mathrm{think}(x, \mathrm{escape}(y))))$

Limitations on scope level

(4) A warden thinks that every prisoner escaped.

$\Rightarrow$ $\textsf{some}(\textsf{warden}, \lambda x.\textsf{think}(x, \textsf{every}(\textsf{prisoner}, \textsf{escape})))$
$\not\Rightarrow$ $\textsf{every}(\textsf{prisoner}, \lambda y.\textsf{some}(\textsf{warden}, \lambda x.\textsf{think}(x, \textsf{escape}(y))))$

- Received wisdom: the finite clause is a **scope island**—no quantifier inside it can take scope outside it.

Limitations on scope level

(4) A warden thinks that every prisoner escaped.

$\Rightarrow$ $\textsf{some}(\textsf{warden}, \lambda x.\textsf{think}(x, \textsf{every}(\textsf{prisoner}, \textsf{escape})))$

$\not\Rightarrow$ $\textsf{every}(\textsf{prisoner}, \lambda y.\textsf{some}(\textsf{warden}, \lambda x.\textsf{think}(x, \textsf{escape}(y))))$

- Received wisdom: the finite clause is a **scope island**—no quantifier inside it can take scope outside it.
- Does not apply to indefinites (maybe they aren't quantifiers?):

(5) Every warden thinks that a prisoner escaped.

$\Rightarrow$ $\textsf{every}(\textsf{warden}, \lambda x.\textsf{think}(x, \textsf{some}(\textsf{prisoner}, \textsf{escape})))$

$\Rightarrow$ $\textsf{some}(\textsf{prisoner}, \lambda y.\textsf{every}(\textsf{warden}, \lambda x.\textsf{think}(x, \textsf{escape}(y))))$

The received wisdom seems to favour the IOFU approach

$$f : \begin{bmatrix} \textsc{pred} & \text{`think}\langle g, h\rangle\text{'} \\ \textsc{tense} & \textsc{pres} \\ \textsc{subj} & g : \left[\text{“a warden”}\right] \\ \textsc{comp} & h : \begin{bmatrix} \textsc{pred} & \text{`escape}\langle i\rangle\text{'} \\ \textsc{tense} & \textsc{past} \\ \textsc{subj} & i : \left[\text{“every prisoner”}\right] \end{bmatrix} \end{bmatrix}$$

The received wisdom seems to favour the IOFU approach

$$f : \begin{bmatrix} \textsc{pred} & \text{`think}\langle g, h\rangle\text{'} \\ \textsc{tense} & \textsc{pres} \\ \textsc{subj} & g : \begin{bmatrix}\text{``a warden''}\end{bmatrix} \\ \textsc{comp} & h : \begin{bmatrix} \textsc{pred} & \text{`escape}\langle i\rangle\text{'} \\ \textsc{tense} & \textsc{past} \\ \textsc{subj} & i : \begin{bmatrix}\text{``every prisoner''}\end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\%A = \begin{pmatrix} \textsc{gf}^{*} & \textsc{gf}\uparrow \\ \neg(\rightarrow \textsc{tense}) & \end{pmatrix}$$

$$\textit{every prisoner} \rightsquigarrow \lambda P.\textbf{every}(\textbf{prisoner}, P) : (\uparrow_e \multimap \%A_p) \multimap \%A_p$$

The received wisdom seems to favour the IOFU approach

$$f : \begin{bmatrix} \textsc{pred} & \text{`think}\langle g, h\rangle\text{'} \\ \textsc{tense} & \textsc{pres} \\ \textsc{subj} & g : \begin{bmatrix} \text{"a warden"} \end{bmatrix} \\ \textsc{comp} & h : \begin{bmatrix} \textsc{pred} & \text{`escape}\langle i\rangle\text{'} \\ \textsc{tense} & \textsc{past} \\ \textsc{subj} & i : \begin{bmatrix} \text{"every prisoner"} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$$\%A = \left(\begin{array}{cc} \textsc{gf}^* & \textsc{gf} \uparrow \\ \neg(\rightarrow \textsc{tense}) & \end{array} \right)$$

$$\textit{every prisoner} \rightsquigarrow \lambda P.\textbf{every}(\textbf{prisoner}, P) : (\uparrow_e \multimap \%A_p) \multimap \%A_p$$

$$\%B = (\textsc{gf}^* \; \textsc{gf} \uparrow)$$

$$\textit{a warden} \rightsquigarrow \lambda P.\textbf{some}(\textbf{warden}, P) : (\uparrow_e \multimap \%B_p) \multimap \%B_p$$

Wrinkles for the received wisdom

Not all finite clauses are scope islands:

(6) An accomplice ensured that every prisoner escaped.

$\Rightarrow$ $\mathsf{some}(\mathsf{accomplice}, \lambda x.\mathsf{ensure}(x, \mathsf{every}(\mathsf{prisoner}, \mathsf{escape})))$

$\Rightarrow$ $\mathsf{every}(\mathsf{prisoner}, \lambda y.\mathsf{some}(\mathsf{accomplice}, \lambda x.\mathsf{ensure}(x, \mathsf{escape}(y))))$

- ‘ensure’ allows some quantifiers to take scope outside the clause it embeds …

Wrinkles for the received wisdom

Not all finite clauses are scope islands:

(6) An accomplice ensured that every prisoner escaped.

$\Rightarrow$ $\textsf{some}(\textsf{accomplice}, \lambda x.\textsf{ensure}(x, \textsf{every}(\textsf{prisoner}, \textsf{escape})))$

$\Rightarrow$ $\textsf{every}(\textsf{prisoner}, \lambda y.\textsf{some}(\textsf{accomplice}, \lambda x.\textsf{ensure}(x, \textsf{escape}(y))))$

- 'ensure' allows some quantifiers to take scope outside the clause it embeds ... but not all of them:

(7) ?An accomplice ensured that no prisoner escaped.

$\Rightarrow$ $\textsf{some}(\textsf{accomplice}, \lambda x.\textsf{ensure}(x, \textsf{not}(\textsf{some}(\textsf{prisoner}, \textsf{escape}))))$

$\not\Rightarrow$ $\textsf{not}(\textsf{some}(\textsf{prisoner}, \lambda y.\textsf{some}(\textsf{accomplice}, \lambda x.\textsf{ensure}(x, \textsf{escape}(y)))))$

Finding a pattern

clause embedder	quantifier *a N*	*every N*	*no N*
think	✓	×	
ensure		✓	×

Finding a pattern

clause embedder	quantifier *a N*	*every N*	*no N*
think	✓	×	×
ensure		✓	×

(8) A warden thinks that no prisoner escaped.

$\Rightarrow$ $\mathsf{some}(\mathsf{warden}, \lambda x.\mathsf{think}(x, \mathsf{not}(\mathsf{some}(\mathsf{prisoner}, \mathsf{escape}))))$

$\not\Rightarrow$ $\mathsf{not}(\mathsf{some}(\mathsf{prisoner}, \lambda y.\mathsf{some}(\mathsf{warden}, \lambda x.\mathsf{think}(x, \mathsf{escape}(y)))))$

Finding a pattern

clause embedder	quantifier a N	$every$ N	no N
think	✓	×	×
ensure	✓	✓	×

(8) A warden thinks that no prisoner escaped.

$\Rightarrow$ $\mathsf{some}(\mathsf{warden}, \lambda x.\mathsf{think}(x, \mathsf{not}(\mathsf{some}(\mathsf{prisoner}, \mathsf{escape}))))$

$\not\Rightarrow$ $\mathsf{not}(\mathsf{some}(\mathsf{prisoner}, \lambda y.\mathsf{some}(\mathsf{warden}, \lambda x.\mathsf{think}(x, \mathsf{escape}(y)))))$

(9) Every accomplice ensured that a prisoner escaped.

$\Rightarrow$ $\mathsf{every}(\mathsf{accomplice}, \lambda x.\mathsf{ensure}(x, \mathsf{some}(\mathsf{prisoner}, \mathsf{escape})))$

$\Rightarrow$ $\mathsf{some}(\mathsf{prisoner}, \lambda y.\mathsf{every}(\mathsf{accomplice}, \lambda x.\mathsf{ensure}(x, \mathsf{escape}(y))))$

The Scope Island Subset Constraint (SISC)

A proposed generalization from Barker (2021):

> *Given any two scope takers, the set of scope islands that trap one is a subset of the set of scope islands that trap the other.*

The Scope Island Subset Constraint (SISC)

A proposed generalization from Barker (2021):

> *Given any two scope takers, the set of scope islands that trap one is a subset of the set of scope islands that trap the other.*

Implies an implicational relationship:

- Being a scope island for *a N* implies being a scope island for *every N*.
- Being a scope island for *every N* implies being a scope island for *no N*.
- Being trapped by *ensure* implies being trapped by *think*.
- ...

Another example

To be licensed, a negative polarity item (NPI) like *any N* must be interpreted within the scope of an appropriate 'negative' licensor—Fry (1999) shows a method for ensuring this in LFG+Glue.

(10) #Anyone will come to the party.

(11) Jim doubts that anyone will come to the party.

(12) Lyn will be happy if anyone comes to the party.

Another example

To be licensed, a negative polarity item (NPI) like *any N* must be interpreted within the scope of an appropriate 'negative' licensor—Fry (1999) shows a method for ensuring this in LFG+Glue.

(10) #Anyone will come to the party.

(11) Jim doubts that anyone will come to the party.

(12) Lyn will be happy if anyone comes to the party.

But where there's more than one licensor available, can an NPI take any licensed scope position?

NPI licensors as scope island projectors

It seems that *any N* **can** take scope out of the complement of *doubt* so long as it's otherwise licensed.

(13) Lyn will be happy if Jim doubts that anyone is coming to the party.

$\Rightarrow$ $\mathsf{if}(\mathsf{doubt}(\mathsf{jim}, \mathsf{someone}(\mathsf{come})), \mathsf{happy}(\mathsf{lyn}))$

$\Rightarrow$ $\mathsf{if}(\mathsf{someone}(\lambda x.\mathsf{doubt}(\mathsf{jim}, \mathsf{come}(x))), \mathsf{happy}(\mathsf{lyn}))$

NPI licensors as scope island projectors

It seems that *any N* **can** take scope out of the complement of *doubt* so long as it's otherwise licensed.

(13) Lyn will be happy if Jim doubts that anyone is coming to the party.

$\Rightarrow$ $\mathsf{if}(\mathsf{doubt}(\mathsf{jim}, \mathsf{someone}(\mathsf{come})), \mathsf{happy}(\mathsf{lyn}))$
$\Rightarrow$ $\mathsf{if}(\mathsf{someone}(\lambda x.\mathsf{doubt}(\mathsf{jim}, \mathsf{come}(x))), \mathsf{happy}(\mathsf{lyn}))$

But it **can't** take scope out of the complement of *if*.

(14) Jim doubts that Lyn will be happy if anyone comes to the party.

$\Rightarrow$ $\mathsf{doubt}(\mathsf{jim}, \mathsf{if}(\mathsf{someone}(\mathsf{come}), \mathsf{happy}(\mathsf{lyn})))$
$\not\Rightarrow$ $\mathsf{doubt}(\mathsf{jim}, \mathsf{someone}(\lambda x.\mathsf{if}(\mathsf{come}(x), \mathsf{happy}(\mathsf{lyn}))))$

Following the pattern

clause	quantifier			
embedder	*an N*	*any N*	*every N*	*no N*
if		×		
think	✓		×	×
doubt		✓		
ensure	✓		✓	×

Following the pattern

clause embedder	quantifier			
	an N	*any N*	*every N*	*no N*
if	✓	×	×	×
think	✓	✓	×	×
doubt	✓	✓	×	×
ensure	✓	✓	✓	×

Following the pattern

clause embedder	*an N*	*any N*	*every N*	*no N*	**island strength**
if	✓	×	×	×	3
think	✓	✓	×	×	2
doubt	✓	✓	×	×	2
ensure	✓	✓	✓	×	1
escaper strength	3	2	1	0	

Approaching the data

Approaching the data

Blocking features and off-path constraints

Different clause types at f-structure

- We can still use constraints on an IOFU path to enforce scope islands.

Different clause types at f-structure

- We can still use constraints on an IOFU path to enforce scope islands.
- But it's not clear that we can tie these to independently-given syntactic features.

Different clause types at f-structure

- We can still use constraints on an IOFU path to enforce scope islands.
- But it's not clear that we can tie these to independently-given syntactic features. We would probably need something like this:

thinks *V*

$(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{0, 1\}$

ensures *V*

$(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{0\}$

thinks $\quad V$

$(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{0, 1\}$

ensures $\quad V$

$(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{0\}$

everyone $\quad N$

$$\%C = \begin{pmatrix} \text{GF}^{*} & \text{GF}\uparrow \\ \neg(1 \in (\rightarrow \textsc{ScopeIsland})) & \end{pmatrix}$$

$\textbf{everyone} : (\uparrow_e \multimap \%C_p) \multimap \%C_p$

no-one $\quad N$

$$\%D = \begin{pmatrix} \text{GF}^{*} & \text{GF}\uparrow \\ \neg(0 \in (\rightarrow \textsc{ScopeIsland})) & \end{pmatrix}$$

$\lambda P.\textbf{not}(\textbf{someone}(P)) : (\uparrow_e \multimap \%D_p) \multimap \%D_p$

Problems

- The SCOPEISLAND feature is not independently motivated.

Problems

- The ScopeIsland feature is not independently motivated.
- There's no obvious way to enforce the SISC.

Problems

- The ScopeIsland feature is not independently motivated.
- There's no obvious way to enforce the SISC. For example, there's nothing to stop a clause-embedder from containing the description $(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{1\}$, allowing *no-one* to take scope out of it but not *everyone*.

Problems

- The ScopeIsland feature is not independently motivated.
- There's no obvious way to enforce the SISC. For example, there's nothing to stop a clause-embedder from containing the description $(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{1\}$, allowing *no-one* to take scope out of it but not *everyone*.
- A completely different theory would be needed for **intra**-clausal scope rigidity, e.g.

(15) Every warden checked no prisoner(s).

$\Rightarrow$ $\textbf{every}(\textbf{warden}, \lambda x.\textbf{not}(\textbf{some}(\textbf{prisoner}, \lambda y.\textbf{check}(x, y))))$

$\not\Rightarrow$ $\textbf{not}(\textbf{some}(\textbf{prisoner}, \lambda y.\textbf{every}(\textbf{warden}, \lambda x.\textbf{check}(x, y))))$

Problems

- The ScopeIsland feature is not independently motivated.
- There's no obvious way to enforce the SISC. For example, there's nothing to stop a clause-embedder from containing the description $(\uparrow \textsc{comp}\ \textsc{ScopeIsland}) = \{1\}$, allowing *no-one* to take scope out of it but not *everyone*.
- A completely different theory would be needed for **intra**-clausal scope rigidity, e.g.

(15) Every warden checked no prisoner(s).

$\Rightarrow$ $\mathbf{every}(\mathbf{warden}, \lambda x.\mathbf{not}(\mathbf{some}(\mathbf{prisoner}, \lambda y.\mathbf{check}(x, y))))$

$\not\Rightarrow$ $\mathbf{not}(\mathbf{some}(\mathbf{prisoner}, \lambda y.\mathbf{every}(\mathbf{warden}, \lambda x.\mathbf{check}(x, y))))$

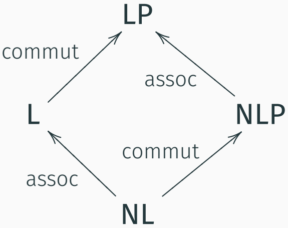
- (It forces us to use IOFU to fix scope level, rather than linear logic quantification.)

Approaching the data

Multi-modal Glue semantics

Properties of linear logic for Glue

The base fragment of linear logic used in Glue is equivalent to the Lambek calculus with permutation or LP, and so relates to other substructural type logics like this:



Properties of linear logic for Glue

The base fragment of linear logic used in Glue is equivalent to the Lambek calculus with permutation or **LP**, and so relates to other substructural type logics like this:

$$
\begin{tikzcd}
 & \mathbf{LP} & \\
\mathbf{L} \arrow[ur, "\text{commut}"] & & \mathbf{NLP} \arrow[ul, "\text{assoc}"'] \\
 & \mathbf{NL} \arrow[ul, "\text{assoc}"] \arrow[ur, "\text{commut}"'] &
\end{tikzcd}
$$

Some properties of **LP**:

Properties of linear logic for Glue

The base fragment of linear logic used in Glue is equivalent to the Lambek calculus with permutation or **LP**, and so relates to other substructural type logics like this:

$$\begin{tikzcd}
& \mathbf{LP} & \\
\mathbf{L} \arrow[ur, "\text{commut}"] & & \mathbf{NLP} \arrow[ul, "\text{assoc}"'] \\
& \mathbf{NL} \arrow[ul, "\text{assoc}"] \arrow[ur, "\text{commut}"'] &
\end{tikzcd}$$

Some properties of **LP**:

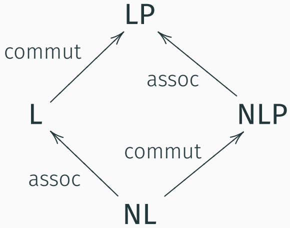
Commutativity

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

The order of premises doesn't matter.

Properties of linear logic for Glue

The base fragment of linear logic used in Glue is equivalent to the Lambek calculus with permutation or **LP**, and so relates to other substructural type logics like this:



Some properties of **LP**:

Commutativity

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

The order of premises doesn't matter.

Associativity

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

The grouping of premises doesn't matter.

Reflections on the logic

- LP has been a good choice of logic for Glue: unlike in categorial grammar, the logic is not meant to account for word order and so it makes sense for it to be commutative.

Reflections on the logic

- LP has been a good choice of logic for Glue: unlike in categorial grammar, the logic is not meant to account for word order and so it makes sense for it to be commutative.
- So far it has also made sense for the logic to be associative, but scope islands may actually give us a reason to care about how premises are grouped, and so restrict associativity.

Reflections on the logic

- LP has been a good choice of logic for Glue: unlike in categorial grammar, the logic is not meant to account for word order and so it makes sense for it to be commutative.
- So far it has also made sense for the logic to be associative, but scope islands may actually give us a reason to care about how premises are grouped, and so restrict associativity.
- We can do so selectively by combining elements of LP (as before) and NLP (which is non-associative) in a multimodal system, where the modes correspond to the island/escaper strengths outlined above.

Proposed rules of inference for multi-modal Glue

$$\frac{}{x : A \vdash x : A}\ \text{axiom}$$

For modes $i, j \in \{\_, \nearrow 1, \nearrow 2, \nearrow 3, \swarrow 1, \swarrow 2, \swarrow 3\}$:

$$\frac{\Gamma \vdash x : A \quad \Delta \vdash f : A \multimap_i B}{(\Gamma, \Delta)^i \vdash f(x) : B} \multimap_i \mathrm{E} \qquad \frac{(x : A, \Gamma)^i \vdash y : B}{\Gamma \vdash \lambda x.y : A \multimap_i B} \multimap_i \mathrm{I}$$

$$\frac{(\Gamma, \Delta)^i \vdash x : A}{(\Delta, \Gamma)^i \vdash x : A}\ \mathrm{P} \qquad \frac{((\Gamma, \Delta)^i, \Sigma)^j \vdash x : A}{(\Gamma, (\Delta, \Sigma)^j)^i \vdash x : A}\ \mathrm{MA}$$

provided that j
does not block i

Comments on the rules

- Because we no longer assume generalized associativity, there is bracketing on the left hand side of sequents.

Comments on the rules

- Because we no longer assume generalized associativity, there is bracketing on the left hand side of sequents.
- The mode indices on those brackets correspond to mode indices on occurrences of $\multimap$.

Comments on the rules

- Because we no longer assume generalized associativity, there is bracketing on the left hand side of sequents.
- The mode indices on those brackets correspond to mode indices on occurrences of $\multimap$.
- Commutativity is ensured by the structural rule P (for *permutation*), and we have restricted associativity thanks to the rule MA (*mixed associativity*).

Comments on the rules

- Because we no longer assume generalized associativity, there is bracketing on the left hand side of sequents.
- The mode indices on those brackets correspond to mode indices on occurrences of $\multimap$.
- Commutativity is ensured by the structural rule P (for *permutation*), and we have restricted associativity thanks to the rule MA (*mixed associativity*).
- MA, in combination with the lexicon, permits just the right scope takers to escape from just the right islands.

Blocking and escaping modalities

if	$\swarrow 3$	*an N*	$\nearrow 3$
think	$\swarrow 2$	*any N*	$\nearrow 2$
ensure	$\swarrow 1$	*every N*	$\nearrow 1$
		no N	$\_$

Mode j blocks mode i iff:

- $j = \swarrow n$ for some n, and
  - $i = \_$, or
  - $i = \nearrow m$ for some $m < n$.

Lexicon

Clause embedders:

$$\textit{if} \rightsquigarrow \lambda p.\lambda q.\mathbf{if}(p,q) : \uparrow_p \multimap_{\swarrow 3} ((\text{ADJ} \in \uparrow)_p \multimap (\text{ADJ} \in \uparrow)_p)$$

$$\textit{thinks} \rightsquigarrow \lambda p.\lambda x.\mathbf{think}(x,p) : (\uparrow \text{COMP})_p \multimap_{\swarrow 2} ((\uparrow \text{SUBJ})_e \multimap_i \uparrow_p)$$

$$\textit{ensured} \rightsquigarrow \lambda p.\lambda x.\mathbf{ensure}(x,p) : (\uparrow \text{COMP})_p \multimap_{\swarrow 1} ((\uparrow \text{SUBJ})_e \multimap_i \uparrow_p)$$

Scope takers:

$$\textit{a} \rightsquigarrow \lambda P.\lambda Q.\mathbf{some}(P,Q) : \forall X.(\uparrow_e \multimap \uparrow_p) \multimap ((\uparrow_e \multimap_{\nearrow 3} X_p) \multimap X_p)$$

$$\textit{any} \rightsquigarrow \lambda P.\lambda Q.\mathbf{some}(P,Q) : \forall X.(\uparrow_e \multimap \uparrow_p) \multimap ((\uparrow_e \multimap_{\nearrow 2} X_p) \multimap X_p)$$

$$\textit{every} \rightsquigarrow \lambda P.\lambda Q.\mathbf{every}(P,Q) : \forall X.(\uparrow_e \multimap \uparrow_p) \multimap ((\uparrow_e \multimap_{\nearrow 1} X_p) \multimap X_p)$$

$$\textit{no} \rightsquigarrow \lambda P.\lambda Q.\mathbf{not}(\mathbf{some}(P,Q)) : \forall X.(\uparrow_e \multimap \uparrow_p) \multimap ((\uparrow_e \multimap X_p) \multimap X_p)$$

- $\multimap$ (with no mode shown) means $\multimap_{-}$.
- $\multimap_i$ means free choice of mode.

(6) An accomplice ensured that every prisoner escaped.

$$f : \begin{bmatrix} \textsc{pred} & \text{`ensure}\langle g, h \rangle\text{'} \\ \textsc{subj} & g : \begin{bmatrix} \text{“an accomplice”} \end{bmatrix} \\ \textsc{comp} & h : \begin{bmatrix} \textsc{pred} & \text{`escape}\langle i \rangle\text{'} \\ \textsc{subj} & i : \begin{bmatrix} \text{“every prisoner”} \end{bmatrix} \end{bmatrix} \end{bmatrix}$$

$[\text{an accomplice}] := \lambda P.\mathbf{some}(\mathbf{accomplice}, P) : (g_e \multimap_{\nearrow 3} f_p) \multimap f_p$

$[\text{ensured}] := \lambda p.\lambda x.\mathbf{ensure}(x, p) : h_p \multimap_{\swarrow 1} (g_e \multimap_{\nearrow 3} f_p)$

$[\text{every prisoner}] := \lambda P.\mathbf{every}(\mathbf{prisoner}, P) : \forall X.(i_e \multimap_{\nearrow 1} X_p) \multimap X_p$

$[\text{escaped}] := \mathbf{escape} : i_e \multimap_{\nearrow 1} h_p$

$$
\dfrac{
  \dfrac{
    \dfrac{
      \begin{array}{c}[\text{escaped}] \vdash \\ \textbf{escape} : \\ i_e \multimap_{\nearrow 1} h_p\end{array}
      \qquad
      \begin{array}{c}\vdots \\ [\text{every prisoner}] \vdash \\ \lambda P.\textbf{every}(\textbf{prisoner}, P) : \\ (i_e \multimap_{\nearrow 1} h_p) \multimap h_p\end{array}
    }{
      \begin{array}{c}([\text{escaped}], [\text{every prisoner}]) \vdash \\ \textbf{every}(\textbf{prisoner}, \textbf{escape}) : h_p\end{array}
    }
    \qquad
    \begin{array}{c}[\text{ensured}] \vdash \\ \lambda p.\lambda x.\textbf{ensure}(x, p) : \\ h_p \multimap_{\swarrow 1} (g_e \multimap_{\nearrow 3} f_p\ )\end{array}
  }{
    \begin{array}{c}(([\text{escaped}], [\text{every prisoner}]), [\text{ensured}])^{\swarrow 1} \vdash \\ \lambda x.\textbf{ensure}(x, \textbf{every}(\textbf{prisoner}, \textbf{escape})) : g_e \multimap_{\swarrow 3} f_p\end{array}
  }
  \qquad
  \begin{array}{c}\vdots \\ [\text{an accomplice}] \vdash \\ \lambda P.\textbf{some}(\textbf{accomplice}, P) : \\ (g_e \multimap_{\nearrow 3} f_p) \multimap f_p\end{array}
}{
  \begin{array}{c}((([\text{escaped}], [\text{every prisoner}]), [\text{ensured}])^{\swarrow 1}, [\text{an accomplice}]) \vdash \\ \textbf{some}(\textbf{accomplice}, \lambda x.\textbf{ensure}(x, \textbf{every}(\textbf{prisoner}, \textbf{escape}))) : f_p\end{array}
}
$$

Beginning inverse scope

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{}{\begin{array}{c} y : i_e \vdash \\ y : i_e \end{array}}
\qquad
\begin{array}{c} [\text{escaped}] \vdash \\ \textbf{escape} : \\ i_e \multimap_{\nearrow 1} h_p \end{array}
}{
\begin{array}{c} (y : i_e, [\text{escaped}])^{\nearrow 1} \vdash \\ \textbf{escape}(y) : h_p \end{array}
}
\qquad
\begin{array}{c} [\text{ensured}] \vdash \\ \lambda p.\lambda x.\textbf{ensure}(x, p) : \\ h_p \multimap_{\swarrow 1} (g_e \multimap_{\nearrow 3} f_p\ ) \end{array}
}{
\begin{array}{c} ((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{ensuped}])^{\swarrow 1} \vdash \\ \lambda x.\textbf{ensure}(x, \textbf{escape}(y)) : g_e \multimap_{\nearrow 3} f_p \end{array}
}
\qquad
\begin{array}{c} \vdots \\ [\text{an accomplice}] \vdash \\ \lambda P.\textbf{some}(\textbf{accomplice}, P) : \\ (g_e \multimap_{\nearrow 3} f_p) \multimap f_p \end{array}
}{
\begin{array}{c} (((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{ensuped}])^{\swarrow 1}, [\text{an accomplice}]) \vdash \\ \textbf{some}(\textbf{accomplice}, \lambda x.\textbf{ensure}(x, \textbf{escape}(y))) : f_p \end{array}
}
$$

Structural rules

$(((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{ensured}])^{\swarrow 1}, [\text{an accomplice}]) \vdash$
$\quad \mathbf{some}(\mathbf{accomplice}, \lambda x.\mathbf{ensure}(x, \mathbf{escape}(y))) : f_p$

We need to ‘move’ *y* to the outside of the structure so it can be abstracted.

$$(((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{ensured}])^{\swarrow 1}, [\text{an accomplice}]) \vdash \\ \quad \mathbf{some}(\mathbf{accomplice}, \lambda x.\mathbf{ensure}(x, \mathbf{escape}(y))) : f_p$$

We need to 'move' y to the outside of the structure so it can be abstracted. This is licit:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}(((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{ensuped}])^{\swarrow 1}, [\text{an accomplice}]) \vdash \\ \mathbf{some}(\mathbf{accomplice}, \lambda x.\mathbf{ensure}(x, \mathbf{escape}(y))) : f_p\end{array}}{\begin{array}{c}((y : i_e, ([\text{escaped}], [\text{ensuped}])^{\swarrow 1})^{\nearrow 1}, [\text{an accomplice}]) \vdash \\ \mathbf{some}(\mathbf{accomplice}, \lambda x.\mathbf{ensure}(x, \mathbf{escape}(y))) : f_p\end{array}}\ \mathrm{MA}}{\begin{array}{c}(y : i_e, (([\text{escaped}], [\text{ensuped}])^{\swarrow 1}, [\text{an accomplice}]))^{\nearrow 1} \vdash \\ \mathbf{some}(\mathbf{accomplice}, \lambda x.\mathbf{ensure}(x, \mathbf{escape}(y))) : f_p\end{array}}\ \mathrm{MA}}{\begin{array}{c}(([\text{escaped}], [\text{ensuped}])^{\swarrow 1}, [\text{an accomplice}]) \vdash \\ \lambda y.\mathbf{some}(\mathbf{accomplice}, \lambda x.\mathbf{ensure}(x, \mathbf{escape}(y))) : i_e \multimap_{\nearrow 1} f_p\end{array}}\ \multimap_{\nearrow 1} \mathrm{I}$$

Inverse scope

$$\dfrac{\begin{array}{c}(([\text{escaped}],[\text{ensured}])^{\swarrow 1},[\text{an accomplice}]) \vdash \\ \lambda y.\mathbf{some}(\mathbf{accomplice},\lambda x.\mathbf{ensure}(x,\mathbf{escape}(y))) : \\ i_e \multimap_{\nearrow 1} f_p\end{array} \quad \begin{array}{c}\vdots \\ [\text{every prisoner}] \vdash \\ \lambda P.\mathbf{every}(\mathbf{prisoner},P) : \\ (i_e \multimap_{\nearrow 1} f_p) \multimap f_p\end{array}}{\begin{array}{c}((([\text{escaped}],[\text{ensured}])^{\swarrow 1},[\text{an accomplice}]),[\text{every prisoner}]) \vdash \\ \mathbf{every}(\mathbf{prisoner},\lambda y.\mathbf{some}(\mathbf{accomplice},\lambda x.\mathbf{ensure}(x,\mathbf{escape}(y)))) : f_p\end{array}}$$

(4) A warden thinks that every prisoner escaped.

Surface scope:

$$\dfrac{\dfrac{\dfrac{\begin{array}{c}[\text{escaped}] \vdash \\ \textbf{escape} : \\ i_e \multimap_{\nearrow 1} h_p\end{array} \qquad \begin{array}{c}\vdots \\ [\text{every prisoner}] \vdash \\ \lambda P.\textbf{every}(\textbf{prisoner}, P) : \\ (i_e \multimap_{\nearrow 1} h_p) \multimap h_p\end{array}}{\begin{array}{c}([\text{escaped}], [\text{every prisoner}]) \vdash \\ \textbf{every}(\textbf{prisoner}, \textbf{escape}) : h_p\end{array}} \qquad \begin{array}{c}[\text{thinks}] \vdash \\ \lambda p.\lambda x.\textbf{think}(x, p) : \\ h_p \multimap_{\swarrow 2} (g_e \multimap_{\nearrow 3} f_p)\end{array}}{\begin{array}{c}(([\text{escaped}], [\text{every prisoner}]), [\text{thinks}])^{\swarrow 2} \vdash \\ \lambda x.\textbf{think}(x, \textbf{every}(\textbf{prisoner}, \textbf{escape})) : g_e \multimap_{\nearrow 3} f_p\end{array}} \qquad \begin{array}{c}\vdots \\ [\text{a warden}] \vdash \\ \lambda P.\textbf{some}(\textbf{warden}, P) : \\ (g_e \multimap_{\nearrow 3} f_p) \multimap f_p\end{array}}{\begin{array}{c}((([\text{escaped}], [\text{every prisoner}]), [\text{thinks}])^{\swarrow 2}, [\text{a warden}]) \vdash \\ \textbf{some}(\textbf{warden}, \lambda x.\textbf{think}(x, \textbf{every}(\textbf{prisoner}, \textbf{escape}))) : f_p\end{array}}$$

Attempting inverse scope

$$
\dfrac{
  \dfrac{
    \dfrac{
      \begin{array}{c} y : i_e \vdash \\ y : i_e \end{array}
      \qquad
      \begin{array}{c} [\text{escaped}] \vdash \\ \textbf{escape} : \\ i_e \multimap_{\nearrow 1} h_p \end{array}
    }{
      \begin{array}{c} (y : i_e, [\text{escaped}])^{\nearrow 1} \vdash \\ \textbf{escape}(y) : h_p \end{array}
    }
    \qquad
    \begin{array}{c} [\text{thinks}] \vdash \\ \lambda p.\lambda x.\textbf{think}(x, p) : \\ h_p \multimap_{\swarrow 2} (g_e \multimap_3 f_p) \end{array}
  }{
    \begin{array}{c} ((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{thinks}])^{\swarrow 2} \vdash \\ \lambda x.\textbf{think}(x, \textbf{escape}(y)) : g_e \multimap_3 f_p \end{array}
  }
  \qquad
  \begin{array}{c} [\text{a warden}] \vdash \\ \lambda P.\textbf{some}(\textbf{warden}, P) : \\ (g_e \multimap_3 f_p) \multimap f_p \end{array}
}{
  \begin{array}{c} (((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{thinks}])^{\swarrow 2}, [\text{a warden}]) \vdash \\ \textbf{some}(\textbf{warden}, \lambda x.\textbf{think}(x, \textbf{escape}(y))) : f_p \end{array}
}
$$

$$\dfrac{\dfrac{\dfrac{\begin{array}{c} y : i_e \vdash \\ y : i_e \end{array} \quad \begin{array}{c} [\text{escaped}] \vdash \\ \mathbf{escape} : \\ i_e \multimap_{\nearrow 1} h_p \end{array}}{\begin{array}{c} (y : i_e, [\text{escaped}])^{\nearrow 1} \vdash \\ \mathbf{escape}(y) : h_p \end{array}} \quad \begin{array}{c} [\text{thinks}] \vdash \\ \lambda p.\lambda x.\mathbf{think}(x, p) : \\ h_p \multimap_{\swarrow 2} (g_e \multimap_3 f_p) \end{array}}{\begin{array}{c} ((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{thinks}])^{\swarrow 2} \vdash \\ \lambda x.\mathbf{think}(x, \mathbf{escape}(y)) : g_e \multimap_3 f_p \end{array}} \quad \begin{array}{c} [\text{a warden}] \vdash \\ \lambda P.\mathbf{some}(\mathbf{warden}, P) : \\ (g_e \multimap_3 f_p) \multimap f_p \end{array}}{\begin{array}{c} (((y : i_e, [\text{escaped}])^{\nearrow 1}, [\text{thinks}])^{\swarrow 2}, [\text{a warden}]) \vdash \\ \mathbf{some}(\mathbf{warden}, \lambda x.\mathbf{think}(x, \mathbf{escape}(y))) : f_p \end{array}}$$

$y : i_e$ is stuck inside the structure. MA is not applicable:

$$((\Gamma, \Delta)^{\nearrow 1}, \Sigma)^{\swarrow 2} \vdash A \qquad \not\vdash \qquad (\Gamma, (\Delta, \Sigma)^{\swarrow 2})^{\nearrow 1} \vdash A$$

So we can't get inverse scope.

Rounding out the SISC

The same thing happens if we have

Rounding out the SISC

The same thing happens if we have *no N* embedded under *ensure*, *think* or *if*

$$((\Gamma, \Delta), \Sigma)^{\checkmark 1/2/3} \vdash A \qquad \not\vdash \qquad (\Gamma, (\Delta, \Sigma)^{\checkmark 1/2/3}) \vdash A,$$

Rounding out the SISC

The same thing happens if we have *no N* embedded under *ensure*, *think* or *if*

$$((\Gamma, \Delta), \Sigma)^{\swarrow 1/2/3} \vdash A \qquad \not\vdash \qquad (\Gamma, (\Delta, \Sigma)^{\swarrow 1/2/3}) \vdash A,$$

every N embedded under *if*

$$((\Gamma, \Delta)^{\nearrow 1}, \Sigma)^{\swarrow 3} \vdash A \qquad \not\vdash \qquad (\Gamma, (\Delta, \Sigma)^{\swarrow 3})^{\nearrow 1} \vdash A,$$

Rounding out the SISC

The same thing happens if we have *no N* embedded under *ensure*, *think* or *if*

$$((\Gamma,\Delta),\Sigma)^{\swarrow 1/2/3} \vdash A \qquad \not\vdash \qquad (\Gamma,(\Delta,\Sigma)^{\swarrow 1/2/3}) \vdash A,$$

every N embedded under *if*

$$((\Gamma,\Delta)^{\nearrow 1},\Sigma)^{\swarrow 3} \vdash A \qquad \not\vdash \qquad (\Gamma,(\Delta,\Sigma)^{\swarrow 3})^{\nearrow 1} \vdash A,$$

or *any N* embedded under *if*

$$((\Gamma,\Delta)^{\nearrow 2},\Sigma)^{\swarrow 3} \vdash A \qquad \not\vdash \qquad (\Gamma,(\Delta,\Sigma)^{\swarrow 3})^{\nearrow 2} \vdash A.$$

So the implicational relationship is enforced by the structural rules for the fragment.

Discussion

Comparing the approaches

- The blocking features-based approach is much more conservative, making use only of established LFG+Glue technology.

Comparing the approaches

- The blocking features-based approach is much more conservative, making use only of established LFG+Glue technology.
- It does make use of features that aren't independently motivated, but that would hardly be unusual.

Comparing the approaches

- The blocking features-based approach is much more conservative, making use only of established LFG+Glue technology.
- It does make use of features that aren't independently motivated, but that would hardly be unusual.
- More troublingly, the SISC has to take the form of a generalization over all lexical entries.

Comparing the approaches

- The blocking features-based approach is much more conservative, making use only of established LFG+Glue technology.
- It does make use of features that aren't independently motivated, but that would hardly be unusual.
- More troublingly, the SISC has to take the form of a generalization over all lexical entries.
- In the multi-modal Glue approach the formulation of the MA rule is itself ad-hoc but, that given, the SISC follows automatically.

Comparing the approaches

- The blocking features-based approach is much more conservative, making use only of established LFG+Glue technology.
- It does make use of features that aren't independently motivated, but that would hardly be unusual.
- More troublingly, the SISC has to take the form of a generalization over all lexical entries.
- In the multi-modal Glue approach the formulation of the MA rule is itself ad-hoc but, that given, the SISC follows automatically.
- To finish, let's look at intra-clausal scope rigidity for further considerations.

(15) Every warden checked no prisoner(s).

$\Rightarrow$ **every(warden**, λx.**not(some(prisoner**, λy.**check(**x, y**))))**

$\not\Rightarrow$ **not(some(prisoner**, λy.**every(warden**, λx.**check(**x, y**))))**

$$f : \begin{bmatrix} \textsc{pred} & \text{`check}\langle g, h\rangle\text{'} \\ \textsc{subj} & g : \begin{bmatrix}\text{"every warden"}\end{bmatrix} \\ \textsc{obj} & h : \begin{bmatrix}\text{"no prisoner"}\end{bmatrix} \end{bmatrix}$$

Because there's no embedded clausal f-structure there's no choice of scope level and hence no way to account for this in the blocking features approach.

NPs as scope island inducers?

At the moment we have

$$\textit{every warden} \rightsquigarrow \lambda Q.\mathsf{every}(\mathsf{warden}, Q) : \forall X.((\uparrow_e \multimap_{\nearrow 1} X_p) \multimap X_p)$$

$$\textit{no prisoner} \rightsquigarrow \lambda Q.\mathsf{not}(\mathsf{some}(\mathsf{prisoner}, Q)) : \forall X.((\uparrow_e \multimap X_p) \multimap X_p)$$

NPs as scope island inducers?

At the moment we have

$$\textit{every warden} \rightsquigarrow \lambda Q.\textsf{every}(\textsf{warden}, Q) : \forall X.((\uparrow_e \multimap_{\nearrow 1} X_p) \multimap X_p)$$

$$\textit{no prisoner} \rightsquigarrow \lambda Q.\textsf{not}(\textsf{some}(\textsf{prisoner}, Q)) : \forall X.((\uparrow_e \multimap X_p) \multimap X_p)$$

We can make *every N* block *no N* from taking scope over it by changing the mode on the second linear logic implication:

$$\textit{every warden} \rightsquigarrow \lambda Q.\textsf{every}(\textsf{warden}, Q) : \forall X.((\uparrow_e \multimap_{\nearrow 1} X_p) \multimap_{\swarrow 1} X_p)$$

$$
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\dfrac{
\begin{array}{c} x : g_e \vdash \\ x : g_e \end{array}
\quad
\dfrac{
\begin{array}{c} y : h_e \vdash \\ y : h_e \end{array}
\quad
\begin{array}{c} [\text{checked}] \vdash \\ \lambda v.\lambda u.\mathbf{check}(u, v) : \\ h_e \multimap (g_e \multimap_{\nearrow 1} f_p) \end{array}
}{
\begin{array}{c} (y : h_e, [\text{checked}]) \vdash \\ \lambda u.\mathbf{check}(u, y) : g_e \multimap_{\nearrow 1} f_p \end{array}
}
}{
\begin{array}{c} (x : g_e, (y : h_e, [\text{checked}]))^{\nearrow 1} \vdash \\ \mathbf{check}(x, y) : f_p \end{array}
}
}{
\begin{array}{c} (y : h_e, (x : g_e, [\text{checked}])^{\nearrow 1}) \vdash \\ \mathbf{check}(x, y) : f_p \end{array}
}\ \text{P,MA}
}{
\begin{array}{c} (x : g_e, [\text{checked}])^{\nearrow 1} \vdash \\ \lambda y.\mathbf{check}(x, y) : h_e \multimap f_p \end{array}
}\ \multimap \text{I}
\quad
\begin{array}{c} \vdots \\ [\text{no prisoner}] \vdash \\ \lambda P.\mathbf{not}(\mathbf{some}(\mathbf{prisoner}, P)) : \\ (h_e \multimap f_p) \multimap f_p \end{array}
}{
\begin{array}{c} ((x : g_e, [\text{checked}])^{\nearrow 1}, [\text{no prisoner}]) \\ \mathbf{not}(\mathbf{some}(\mathbf{prisoner}, \lambda y.\mathbf{check}(x, y))) \end{array}
}
}{
\begin{array}{c} (x : g_e, ([\text{checked}], [\text{no prisoner}]))^{\nearrow 1} \\ \mathbf{not}(\mathbf{some}(\mathbf{prisoner}, \lambda y.\mathbf{check}(x, y))) : f_p \end{array}
}\ \text{MA}
}{
\begin{array}{c} ([\text{checked}], [\text{no prisoner}]) \\ \lambda x.\mathbf{not}(\mathbf{some}(\mathbf{prisoner}, \lambda y.\mathbf{check}(x, y))) : g_e \multimap_{\nearrow 1} f_p \end{array}
}\ \multimap_{\nearrow 1} \text{I}
\quad
\begin{array}{c} \vdots \\ [\text{every warden}] \vdash \\ \lambda P.\mathbf{every}(\mathbf{warden}, P) : \\ (g_e \multimap_{\nearrow 1} f_p) \multimap_{\swarrow 1} f_p \end{array}
}{
\begin{array}{c} (([\text{checked}], [\text{no prisoner}]), [\text{every warden}])^{\swarrow 1} \vdash \\ \mathbf{every}(\mathbf{warden}, \lambda x.\mathbf{not}(\mathbf{some}(\mathbf{prisoner}, \lambda y.\mathbf{check}(x, y)))) : f_p \end{array}
}
$$

Attempting inverse scope

$$
\dfrac{
\dfrac{
\begin{array}{c} y : h_e \vdash \\ y : h_e \end{array}
\quad
\begin{array}{c} [\text{checked}] \vdash \\ \lambda v.\lambda u.\mathbf{check}(u, v) : \\ h_e \multimap (g_e \multimap_{\nearrow 1} f_p) \end{array}
}{
\begin{array}{c} (y : h_e, [\text{checked}]) \vdash \\ \lambda u.\mathbf{check}(u, y) : g_e \multimap_{\nearrow 1} f_p \end{array}
}
\qquad
\begin{array}{c} \vdots \\ [\text{every warden}] \vdash \\ \lambda P.\mathbf{every}(\mathbf{warden}, P) : \\ (g_e \multimap_{\nearrow 1} f_p) \multimap_{\swarrow 1} f_p \end{array}
}{
\begin{array}{c} ((y : h_e, [\text{checked}]), [\text{every warden}])^{\swarrow 1} \vdash \\ \mathbf{every}(\mathbf{warden}, \lambda u.\mathbf{check}(u, y)) : f_p \end{array}
}
$$

$$
\dfrac{
  \dfrac{
    \begin{array}{c} y : h_e \vdash \\ y : h_e \end{array}
    \qquad
    \begin{array}{c} [\text{checked}] \vdash \\ \lambda v.\lambda u.\mathbf{check}(u, v) : \\ h_e \multimap (g_e \multimap_{\nearrow 1} f_p) \end{array}
  }{
    \begin{array}{c} (y : h_e, [\text{checked}]) \vdash \\ \lambda u.\mathbf{check}(u, y) : g_e \multimap_{\nearrow 1} f_p \end{array}
  }
  \qquad
  \begin{array}{c} \vdots \\ [\text{every warden}] \vdash \\ \lambda P.\mathbf{every}(\mathbf{warden}, P) : \\ (g_e \multimap_{\nearrow 1} f_p) \multimap_{\swarrow 1} f_p \end{array}
}{
  \begin{array}{c} ((y : h_e, [\text{checked}]), [\text{every warden}])^{\swarrow 1} \vdash \\ \mathbf{every}(\mathbf{warden}, \lambda u.\mathbf{check}(u, y)) : f_p \end{array}
}
$$

- $y : h_e$ is now trapped by the $\swarrow 1$ bracket, so it can't 'move' to the outside of the structure for abstraction.

$$\dfrac{\dfrac{\begin{array}{c} y : h_e \vdash \\ y : h_e \end{array} \quad \begin{array}{c} [\text{checked}] \vdash \\ \lambda v.\lambda u.\mathbf{check}(u, v) : \\ h_e \multimap (g_e \multimap_{\nearrow 1} f_p) \end{array}}{\begin{array}{c} (y : h_e, [\text{checked}]) \vdash \\ \lambda u.\mathbf{check}(u, y) : g_e \multimap_{\nearrow 1} f_p \end{array}} \quad \begin{array}{c} \vdots \\ [\text{every warden}] \vdash \\ \lambda P.\mathbf{every}(\mathbf{warden}, P) : \\ (g_e \multimap_{\nearrow 1} f_p) \multimap_{\swarrow 1} f_p \end{array}}{\begin{array}{c} ((y : h_e, [\text{checked}]), [\text{every warden}])^{\swarrow 1} \vdash \\ \mathbf{every}(\mathbf{warden}, \lambda u.\mathbf{check}(u, y)) : f_p \end{array}}$$

- $y : h_e$ is now trapped by the $\swarrow 1$ bracket, so it can't 'move' to the outside of the structure for abstraction.
- Therefore, inverse scope is impossible.

The problem with NPs as scope island inducers

The proposal just considered would also block the *surface scope* interpretation in a sentence like (16)

(16) No warden checked every prisoner.

by creating the structure

$$((x : g_e, [\text{checked}]), [\text{every prisoner}])^{\surd 1}$$

from which $x : g_e$ would not be able to escape for abstraction.

Avenues for dealing with the problem

At the moment the (non-_) modes keep track of

- blocking vs. escaping: $\swarrow$ vs. $\nearrow$, and
- strength thereof: 1–3.

Avenues for dealing with the problem

At the moment the (non-_) modes keep track of

- blocking vs. escaping: $\swarrow$ vs. $\nearrow$, and
- strength thereof: 1–3.

To enforce intra-clausal scope rigidity by using NPs as island inducers, the modes might also have to keep track of

- argument structure,
- linear order, or
- c-structure embeddedness?

Avenues for dealing with the problem

At the moment the (non-_) modes keep track of

- blocking vs. escaping: $\swarrow$ vs. $\nearrow$, and
- strength thereof: 1–3.

To enforce intra-clausal scope rigidity by using NPs as island inducers, the modes might also have to keep track of

- argument structure,
- linear order, or
- c-structure embeddedness?

This might be too much cateogorial grammar in LFG for many people's tastes, but either way the question of how to enforce (intra- and extra-clausal) scope rigidity in LFG+Glue remains very much open.

Thanks!

This research is funded by the

LEVERHULME TRUST

References

See the accompanying handout.