Quantifiers as Modifiers

Ling 233B 2/4/02

Overview

1. Review of quantifier scope ambiguity
   - Simple cases: *Everyone saw something*
   - Scoping across clauses: *Everyone believed someone yawned*
   - Modified NPs: *Every representative of a company saw a sample*
   - Relation to Cooper storage
   - Scope constraints
2. Quantifiers as modifiers (Gupta & Lamping)
   - Defining modifiers
   - Skeletons and modifiers
   - Quantifiers as skeleton + modifier
   - Horn clause compilation
   - Underspecified and packed inference

Everyone saw something

saw: $a_e \multimap (b_e \multimap c_t)$
everyone: $\forall S_t.\ (a_e \multimap S_t) \multimap S_t$
something: $\forall R_t.\ (b_e \multimap R_t) \multimap R_t$

$$a_e \multimap (b_e \multimap c_t) \quad \equiv \quad b_e \multimap (a_e \multimap c_t)$$

a outscopes b

1.	$a_e \multimap (b_e \multimap c_t)$	
2.	$(b_e \multimap c_t) \multimap c_t$	$R_t = c_t$
3.	$a_e \multimap c_t$	1, 2
4.	$(a_e \multimap c_t) \multimap c_t$	$S_t = c_t$
5.	c_t	3, 4

b outscopes a

1.	$b_e \multimap (a_e \multimap c_t)$	
2.	$(a_e \multimap c_t) \multimap c_t$	$S_t = c_t$
3.	$b_e \multimap c_t$	1, 2
4.	$(b_e \multimap c_t) \multimap c_t$	$R_t = c_t$
5.	c_t	3, 4

Everyone believes something yawned

believe: $a_e \multimap (g_t \multimap f_t)$
everyone: $\forall S_t.\ (a_e \multimap S_t) \multimap S_t$
something: $\forall R_t.\ (b_e \multimap R_t) \multimap R_t$
yawn: $b_e \multimap g_t$

Subordinate clause scope

1.	$b_e \multimap g_t$	
2.	$(b_e \multimap g_t) \multimap g_t$	$R_t = g_t$
3.	g_t	1,2
4.	$a_e \multimap (g_t \multimap f_t)$	
5.	$a_e \multimap f_t$	3, 4
6.	$(a_e \multimap f_t) \multimap f_t$	$S_t = f_t$
7.	f_t	5,6

Main clause scope

1.	$b_e \multimap g_t$	
2.	$[b_e]$	assumption
3.	g_t	1,2
4.	$a_e \multimap (g_t \multimap f_t)$	
5.	$a_e \multimap f_t$	3, 4
6.	$(a_e \multimap f_t) \multimap f_t$	$S_t = f_t$
7.	f_t	5, 6
8.	$b_e \multimap f_t$	discharge 2
9.	$(b_e \multimap g_t) \multimap g_t$	$R_t = f_t$
10.	f_t	

Complex NPs
Every representative of a company saw a sample

Standard example to test correctness of theory of scope

Simple permutation of 3 quantifiers would suggest 3! = 6 scopings

But there are only 5: cannot have rep>sample>company

Missing scope has free variables and vacuous quantification

$$\forall r.\ rep\_of(r, \underline{c}) \rightarrow$$
$$\exists s.\ sample(s) \wedge$$
$$\exists c.\ company(c) \wedge \underline{see(r, s)}$$

Every representative of a company saw a sample

every-rep	:	$pp_t \multimap (r_e \multimap X_t) \multimap X_t$
of	:	$c_e \multimap pp_t$
a-company	:	$(c_e \multimap Y_t) \multimap Y_t$
saw	:	$r_e \multimap s_e \multimap f_t$
a-sample	:	$(s_e \multimap Z_t) \multimap Z_t$

Narrow scope c_e

$c_e \multimap pp_t$
$(c_e \multimap Y_t) \multimap Y_t$
pp_t
$pp_t \multimap (r_e \multimap X_t) \multimap X_t$
$(r_e \multimap X_t) \multimap X_t$
$s_e \multimap (r_e \multimap f_t)$
$s_e \multimap f_t$
$(s_e \multimap Z_t) \multimap Z_t$
f_t

Wide scope c_e

$c_e \multimap pp_t$
$[c_e]$
pp_t
$pp_t \multimap (r_e \multimap X_t) \multimap X_t$
$(r_e \multimap X_t) \multimap X_t$
$s_e \multimap (r_e \multimap f_t)$
$s_e \multimap f_t$
$(s_e \multimap Z_t) \multimap Z_t$
f_t
$c_e \multimap f_t$
$(c_e \multimap Y_t) \multimap Y_t$
f_t

Impossible scope

1.	$c_e \multimap pp_t$	
2.	$[c_e]$	
3.	pp_t	
4.	$pp_t \multimap (r_e \multimap X_t) \multimap X_t$	
5.	$(r_e \multimap X_t) \multimap X_t$	
6.	$r_e \multimap (s_e \multimap f_t)$	scope c_e
7.	$(c_e \multimap Y_t) \multimap Y_t$	$Y_t = f_t$
X.	$r_e \multimap (s_e \multimap f_t)$	INVALID
9.	$(s_e \multimap Z_t) \multimap Z_t$	scope s_e
10.	$r_e \multimap f_t$	
11.	f_t	scope r_e: 5, 10
X.	(2) undischarged	

Cooper Storage

Mechanism to generate quantifier scopings

- Meanings a pair of: ⟨ Main meaning, Quantifier store⟩
- Operation 1: Store a quantifier
  - Give quantified NP a variable meaning, x, in main meaning
  - Place quantifier $\langle Q, x\rangle$ in store
- Operation 2: Retrieve a quantifier
  - Remove $\langle Q, x\rangle$ from storage
  - Bind variable x in main meaning by Q
- Final meaning must have empty quantifier store
- Complex constraints on retrieval (nested Cooper storage) to avoid free variables & vacuous quantification

Glue account of scope is rational reconstruction of Cooper storage

Cooper Storage v. Glue
Everyone saw something

Cooper Storage

Meaning	Store	
$\lambda X, Y.see(X, Y)$	[]	
$\lambda Y.see(x, Y)$	$[\langle\forall, x\rangle]$	store x
$see(x, y)$	$[\langle\forall, x\rangle, \langle\exists, y\rangle]$	store y
$\forall x.see(x, y)$	$[\langle\exists, y\rangle]$	retrieve x
$\exists y.\forall x.see(x, y)$	[]	retrieve y

Glue

$$\dfrac{\dfrac{\dfrac{\dfrac{g \multimap (h \multimap f) \quad [g]^1}{h \multimap f}\,store \quad [h]^2}{f}}{g \multimap f}\,\multimap_{\mathcal{I},1} \quad (g \multimap f) \multimap f}{\dfrac{\dfrac{f}{h \multimap f}\,\multimap_{\mathcal{I},2} \quad (h \multimap f) \multimap f}{f}\,ret}\,ret$$

- Storage = Assumption introduction
- Retrieval = Assumption discharge
- Proper assumption discharge accounts for nested storage

Glue require no scoping mechanism beyond standard deduction rules

Expressing Scope Constraints

Can express scope constraints as constraints on structure of glue derivations (Crouch & van Genabith)

NP h outscopes NP g:

- Last occurence of h in (normal form) derivation must occur below last occurrence of g

e.g.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{g \multimap (h \multimap f) \quad [g]^1}{h \multimap f} \quad [h]^2}{f}}{g \multimap f}\,{\multimap}_{\mathcal{I},1} \quad (g \multimap f) \multimap f}{f}}{h \multimap f}\,{\multimap}_{\mathcal{I},2} \quad (h \multimap f) \multimap f}{f}$$

Quantifiers as Modifiers

- Defining modifiers
- Skeletons and modifiers
- Quantifiers as skeleton + modifier
- Horn clause compilation
- Underspecified and packed inference

Problems with Defining Glue Modifiers

Previous class:

- Modifiers as $\phi \multimap \phi$ identities
- Ambiguity through different ways of permuting $\phi \multimap \phi$ around ϕ skeleton

But over-simplistic to define a glue modifier as a logical identity of the form $\phi \multimap \phi$

For example

$$[(a \multimap b) \multimap (a \multimap b)] \;\equiv\; [a \multimap ((a \multimap b) \multimap b)]$$

Want to treat both formulas as modifiers

Also, want to generalize definition of modifier so that quantifiers

$$(g \multimap X) \multimap X$$

can be seen as (dependent) modifiers

Polarity in Glue Formulas

Can classify all linear logic subformulae by polarity:

- Positive: production of resources
- Negative: consumption of resources

Polarity rules for implication

- $(A \multimap B)^+ \implies A^- \multimap^+ B^+$
  $(A \multimap B)^- \implies A^+ \multimap^- B^-$

Atomic polarities for some positive formulas

- $a^- \multimap^+ (b^- \multimap^+ c^+)$
  $a^- \multimap^+ a^+$
  $(a^+ \multimap^- b^-) \multimap^+ (a^- \multimap^+ b^+)$
  $a^- \multimap^+ ((a^+ \multimap^- b^-) \multimap^+ b^+)$
  $(g^+ \multimap^- X^-) \multimap^+ X^+$

Skeleton & Modifier Occurrences

Divide the atomic subformulas in a (positive) formula into modifier and skeleton occurrences

- A positive/negative atomic subformula is a modifier occurrence iff
  - The formula contains a corresponding negative/positive atomic subformula, and
  - The connective linking the two subformulas containing these occurrences is a positive implication
- An atomic subformula is a skeleton occurrence iff it is not a modifier occurrence

Formulas with modifier occurrences underlined

- $a^- \multimap^+ (b^- \multimap^+ c^+)$
- $\underline{a^-} \multimap^+ \underline{a^+}$
- $(\underline{a^+} \multimap^- \underline{b^-}) \multimap^+ (\underline{a^-} \multimap^+ \underline{b^+})$
- $\underline{a^-} \multimap^+ ((\underline{a^+} \multimap^- \underline{b^-}) \multimap^+ \underline{b^+})$
- $(g^+ \multimap^- \underline{X^-}) \multimap^+ \underline{X^+}$
- $X^- \multimap^+ (g^- \multimap X^+)$

Pure and Impure Modifiers

- Pure modifier:
  Formula contains only modifier occurrences
  e.g. $(a \multimap b) \multimap (a \multimap b)$
- Impure modifier:
  Formula contains some skeleton and some modifier occurrences
  e.g. $g \multimap (a \multimap a)$, $a \multimap (g \multimap a)$, $(g \multimap a) \multimap a$
- Purifiable modifier:
  An impure modifier that can be rearranged so that there is just one sub-formula containing all the modifier occurrences and no skeleton occurrences
  e.g. $g \multimap (a \multimap a)$, $a \multimap (g \multimap a)$

  Note: purifiable modifiers can be rearranged to give expressions of the form $skel_1 \multimap (\ldots \multimap (skel_n \multimap pure\_mod)) \ldots)$

Note that quantifiers such as $(g \multimap a) \multimap a$ are not purifiable.

Proofs as Matching Polarities

Linear logic proofs match positive atoms (producers of resources) with negative atoms (consumers of resources).

Suppose all your premises were either pure skeleton, pure modifier or (rearranged) purifiable modifier.

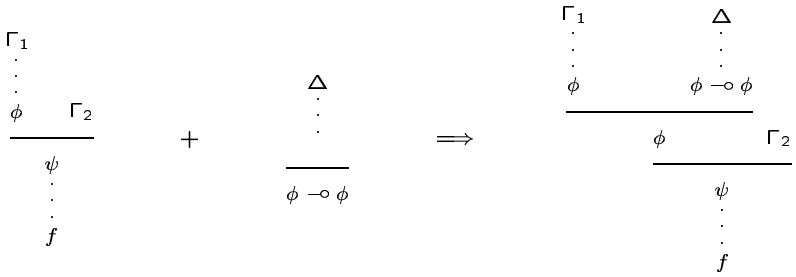
Two stage derivation:

- Initial derivation separating modifiers from skeleton

$$\begin{array}{l} g \multimap h \multimap f \\ a \multimap (f \multimap f) \\ b \multimap (h \multimap f) \multimap (h \multimap f) \\ g,\ h,\ a,\ b \end{array} \quad \Rightarrow \quad \begin{array}{ll} \dfrac{h \quad \dfrac{g \quad g \multimap h \multimap f}{h \multimap f}}{f} & \text{skeleton} \\[2ex] \dfrac{a \quad a \multimap (f \multimap f)}{f \multimap f} & \text{pure modifier} \\[2ex] \dfrac{b \quad b \multimap ((h \multimap f) \multimap (h \multimap f))}{(h \multimap f) \multimap (h \multimap f)} & \text{pure modifier} \end{array}$$

- Final derivation inserts modifiers

Inserting modifiers



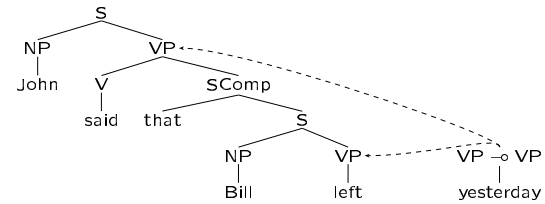
Determine if there is a valid proof while ignoring pure modifiers

Inserting a pure modifier can't make a valid proof invalid

Efficiency Considerations

1. Linguistic observation: nearly always there is just one + and one − skeleton occurence of each semantic resource—Linear time skeleton matching algorithm

2. Modifier insertion allows efficient structure sharing



- Skeleton-modifier approach:
  Skeleton structure built just once
- Chart approach: disjointness conditions on string spans
  Skeleton structure has to be built twice
  — once with low modifier attachment, once without

Problems with Non-Purifiable Modifiers

Non-purifiable modifiers are a problem for the preceding efficient, 2-stage derivation strategy.

We cannot separate a formula like $(g \multimap X) \multimap X$ into a skeleton prefix and a modifier suffix.

Cannot straightforwardly carry out first stage of matching skeletons.
— i.e. cannot match skeleton prefix to get pure modifier

Some trick is needed!

Non-Purifiable = Dependent

A quantifier, like $(g \multimap X) \multimap X$, is a *dependent* (clausal) modifier.

It modifies an X that is missing / dependent on a g, to give an X where the dependency has been discharged.

(The g has as a free variable meaning x that gets bound when the dependency is discharged)

Horn Clauses
Purifying the Impurifiable

Compilation technique (due to Mark Hepple and to Gupta & Lamping) that can separate skeleton from modifier even in non-purifiable modifiers.

Non-purifiable modifiers are non-Horn

Horn clause: $(a_1 \multimap (a_2 \multimap \ldots \multimap (a_n \multimap c)\ldots))$
— Atom, or implication from an atom to a Horn clause.

Horn clauses make for efficient deduction.

Non-Horn clause: $(g \multimap X) \multimap X)$

If we could compile impure modifiers down to Horn clauses, we could do 2-stage derivation

Horn Clause Compilation

Compile non-Horn formula $(a \multimap b) \multimap c$
to pair of Horn formulas $a^i, \quad b\{a^i\} \multimap c$

where

- a^i (uniquely indexed) assumption of a
- $b\{a^i\} \multimap c$ is just $b \multimap c$ — *except that* a^i *must be used in deriving* b

Example

$$a \multimap b \implies a \multimap b$$
$$(a \multimap b) \multimap c \implies a^i,\ b\{a^i\} \multimap c$$

$$\dfrac{\dfrac{a^i \qquad a \multimap b}{b} \qquad b\{a^i\} \multimap c}{c}$$

Proof Rule

$$\dfrac{\phi : b\{a^i\} \multimap c \qquad \begin{array}{c} X{:}a^i \\ \vdots \\ \psi : b \end{array}}{\phi(\lambda X.\psi) : c}$$

Partial Horn-Clause Compilation
Lamping & Gupta 98

Full Horn compilation recursive on subformulas

Partial compilation halts recursion early on identities,
$comp(\phi \multimap \phi) = \phi \multimap \phi$

e.g. $comp(a \multimap ((b \multimap c) \multimap (b \multimap c))) = a \multimap ((b \multimap c) \multimap (b \multimap c))$

<u>Further 'skeletal uniqueness' assumption</u>
After partial compilation, and excluding identity subformulae, entire premise set contains exactly one +ve and one −ve occurrence of each (skeletal) atomic literal.

Linguistically natural restriction on expressiveness.

Separating Skeletons and Modifiers

Given partial Horn compilation & skeletal uniqueness
separate skeleton and modifier trees as follows:

- Hepple-style resolution to build derivation trees.
- But, any tree deriving an identity held out as a modifier.
- Resolution results in one skeleton tree and a set of pure modifier trees (typically in linear time)

Example

$someone : (g \multimap H) \multimap H$ in Horn form is

$$X : g^1, \quad someone : H\{g^1\} \multimap H$$

plus

$$slept : g \multimap f$$

Skeleton Modifier Result

$$\frac{g^1 \quad g \multimap f}{f} \quad + \quad H\{g^1\} \multimap H \quad \Longrightarrow \quad \frac{\dfrac{g^1 \quad g \multimap f}{f} \qquad H\{g^1\} \multimap H}{f}$$

Note that insertion of dependent modifiers must make non-local check on derivation tree to ensure that dependency has actually been used.

Underspecification & Packing

Lamping & Gupta propose underspecified derivations

- Only do step 1 of 2-stage derivation, to separate out skeletons and modifiers
- Do not attempt actual insertion of modifiers

Packed derivations

- Attempt stage 2: insertion of modifiers
- But use structure sharing to compute all possibilities efficiently in terms of space and time

Class implementation:

- Uses Hepple's Horn clause compilation
- But does not do anything clever with modifiers (IP issues)

Summary

1. Review of quantifier scope ambiguity
   - Simple cases: *Everyone saw something*
   - Scoping across clauses: *Everyone believed someone yawned*
   - Modified NPs: *Every representative of a company saw a sample*
   - Relation to Cooper storage
   - Scope constraints

2. Quantifiers as modifiers (Gupta & Lamping)
   - Defining modifiers
   - Skeletons and modifiers
   - Quantifiers as skeleton + modifier
   - Horn clause compilation
   - Underspecified and packed inference