

Questions: Semantic and Computational Issues

Veneeta Dayal and Chung-chieh Shan

October 18, 2006

1. Quantification and binding in functional questions

Goal puzzle:

- (1) a. Who is {every Englishman's/#his} best friend? His mother.
- b. {Every Englishman's/#His} best friend is his mother.
- c. The woman that {every Englishman/#he} loves is his mother.

Transitivity fails:

- (2) a. Who is every Englishman's trusted confidant? His best friend.
- b. Who is every Englishman's trusted confidant? His mother.
- c. #His best friend is his mother.

Specificational copula (“is”) seems to equate some functions (such as the function that maps every Englishman to his best friend) but not others (such as the mother function). Perhaps functions can only be equated given a restricted domain: all Englishmen in this possible world, not just all individuals in any possible world.

Thus let us examine (domains and ranges in) quantification, questions, and binding.

2. Chierchia 1993: “Questions and quantifiers interact in complex ways”

Mainly consider the interaction between a singular wh-phrase and a quantifier.

| | Weak crossover | Quantifier kind | |
|-------------------|----------------|-----------------------|--------------------------|
| | | <i>two (?), every</i> | <i>no, at most three</i> |
| Single answer | ✓ | ✓ | ✓ |
| Functional answer | * | ✓ | ✓ |
| Pair-list answer | * | ✓ | * |

2.1. Three kinds of questions

Weak crossover

- (3) a. Which woman does {every/no} Englishman love?
- b. Which woman loves {every/no} Englishman? (single answer only)

- (4) a. Which dish did every student bring?
 b. Which student brought every dish? (single answer only)

Quantifier kind

- (5) a. Which woman does no Italian married man like?
 b. His mother-in-law
 c. *Giovanni, Maria; Paolo, Francesca; ...
- (6) a. Which requirement do at most three students like?
 b. Their exotic language requirement
 c. *Paul the semantics requirement, Mary the phonology requirement, (and Bill the exotic language requirement)
- (7) Which book does everyone like?
- (8) Which book do two people like?

2.2. Skolem functions

The meaning of functional questions

- (9) Which woman does every Englishman_j love t_i^j ?

$$\lambda p. \exists f_{(e,e)}. [\forall x. [\text{Englishman}(x) \Rightarrow \text{woman}(f(x))]]$$

$$\wedge p = \neg \forall x. [\text{Englishman}(x) \Rightarrow \text{love}(x, f(x))]]$$

Domain: Englishmen. Range: women.

Minimal witness set (required for pair-list questions)

- (10) A minimal witness set for a quantifier \mathcal{P} is a set A such that $A \in \mathcal{P}$ and for no $B \subsetneq A, B \in \mathcal{P}$. Notation: $W(\mathcal{P}, A)$.

Intuition: first pick two people (or first “pick” everyone), then tell me which book each of them likes. “Each quantifier has one or more minimal witness sets”!?

2.3. The meaning of pair-list questions

Two ways to assign denotations to questions

- (11) A set of simple questions
 $\lambda Q. \exists A. [W(\text{everyone}, A) \wedge Q = \lambda p. \exists x y. [A(x) \wedge \text{book}(y) \wedge p = \neg(x \text{ likes } y)]]$
 (here Q is a simple question)
- (12) A lifted simple question (a generalized quantifier over simple questions)
 $\lambda P. \exists A. [W(\text{everyone}, A) \wedge P(\lambda p. \exists x y. [A(x) \wedge \text{book}(y) \wedge p = \neg(x \text{ likes } y)])]$
 (here P is a set of simple questions)

The meaning of pair-list questions

- (13) Which book_i does everyone_j like t_i^j ?

(14) Absorption yields a pair-list question.

which book_i + everyone_j + t_j likes t_i^j \implies

$\lambda P. \exists A. [W(\text{everyone}, A)$
 $\wedge P(\lambda p. \exists x f. [A(x) \wedge \forall x. [A(x) \implies \text{book}(f(x))] \wedge p = \neg(x \text{ likes } f(x))])]$

Absorption is available when the quantifier_j binds into the wh-trace_i^j. Hence the parallel to standard crossover, no matter where the quantifier takes scope.

(15) “we might as well use [quantification over Skolem functions]”

“May’s assumptions [e.g., the Scope Principle] are largely construction specific”

3. Quantifying into question acts

Manfred Krifka. 2001. [Quantifying into question acts](#). *Natural Language Semantics* 9(1):1–40.

Here we show a simplistic polymorphic denotation for *everyone* that generates a pair-list reading for *Which book does everyone like?*. The resulting denotation for a pair-list question is just like an *n*-wh question meaning, where *n* is the number of individuals quantified over. This analysis can be made to account for crossover, but it is unclear why pair-list questions are limited mostly to universal quantifiers.

We will use the `and` function defined in Haskell’s Prelude.

```
and :: [Bool] -> Bool
and []      = True
and (b:bs) = b && and bs
```

Recall also the type-class of monads in the Prelude, from October 4.

```
class Monad m where
  return :: a -> m a
  (>>=)  :: m a -> (a -> m b) -> m b
```

We will use two combinators defined in Haskell’s `Monad` module.

```
liftM :: (Monad m) => (a -> r) -> m a -> m r
liftM f m = m >>= \x -> return (f x)
```

```
mapM :: Monad m => (a -> m b) -> [a] -> m [b]
mapM f []      = return []
mapM f (x:xs) = f x >>= \y ->
  mapM f xs >>= \ys ->
  return (y:ys)
```

(Can you draw a wire diagram for `liftM`? What about for `mapM`?)

3.1. Wh-questioning as a side effect

Recall the `Suspend` monad from October 4:

```
data Suspend a = Done a | Ask (E -> Suspend a)
instance Monad Suspend where
  return a      = Done a
  Done a >>= k = k a
  Ask f >>= k  = Ask (\e -> f e >>= k)
ask :: Suspend E = Ask Done
```

For brevity, we abbreviate Alice, Bob, and Carol to their initials.

```
data E = A | B | C
  deriving (Eq, Ord, Enum, Bounded, Show, Read)
instance Finite E where
  everything = enumEverything
  cardinality = enumCardinality
instance Pretty E
```

(16) Who saw who?

(17) Who is who?

```
> pretty (ask >>= \x ->
          ask >>= \y ->
          return (x == y))
Ask {A: Ask {A: Done True, B: Done False, C: Done False},
     B: Ask {A: Done False, B: Done True, C: Done False},
     C: Ask {A: Done False, B: Done False, C: Done True}}
```

(18) Who went to the store and bought what?

```
> pretty (ask >>= \x ->
          if x == A then return True else ask >>= \y ->
          return (x == y))
Ask {A: Done True,
     B: Ask {A: Done False, B: Done True, C: Done False},
     C: Ask {A: Done False, B: Done False, C: Done True}}
```

3.2. Quantification as another side effect

Type-lifting to generalized quantifiers is also known as the *continuation* monad.

```

data Quantify w a = Quantify ((a -> w) -> w)
instance Monad (Quantify w) where
  return a          = Quantify (\c -> c a)
  Quantify m >>= k = Quantify (\c -> m (\a ->
                                case k a of Quantify n -> n c))

```

We can define the denotation of *everyone* (ignoring animacy) in this monad. Here everything is a list of all individuals; it is a member of the `Finite` type-class.

```

everyone :: Quantify Bool E
everyone = Quantify (\c -> and (map c everything))

```

Quantifiers like *everyone* take scope at a clausal boundary.

```

eval :: Quantify w w -> w
eval (Quantify m) = m id

```

(19) Everyone is Alice.

```

> eval (everyone >>= \x -> return (x == A))
False

```

3.3. Combining two side effects

Monad transformers are one popular way to combine side effects (here, *wh*-questioning and quantification). If *m* is a monad and *t* is a monad transformer, then *t m* is another monad.

```

class MonadT t where
  lift :: (Monad m) => m a -> t m a

```

To mix quantification into *wh*-questioning, we generalize `Quantify` above to a monad transformer `QuantifyT`.

```

data QuantifyT w m a = QuantifyT ((a -> m w) -> m w)
instance Monad (QuantifyT w m) where
  return a          = QuantifyT (\c -> c a)
  QuantifyT m >>= k = QuantifyT (\c -> m (\a ->
                                case k a of QuantifyT n -> n c))
instance MonadT (QuantifyT w) where
  lift m = QuantifyT (\c -> m >>= c)

```

(20) Everyone is Alice.

```

everyoneT :: (Monad m) => QuantifyT Bool m E
everyoneT = QuantifyT (\c -> liftM and (mapM c everything))
evalT :: (Monad m) => QuantifyT w m w -> m w
evalT (QuantifyT m) = m return

```

```

> evalT (everyoneT >>= \x -> return (x == A)) :: [Bool]
[False]

```

Now for the punchline: if the quantifier `everyoneT` “takes scope over” (i.e., evaluates before) the `wh`-word `ask`, then the pair-list reading results!

(21) Who is everyone (equal to)?

```

> pretty (evalT (everyoneT >>= \x ->
                lift ask >>= \y ->
                return (x == y)))
Ask {A:
  Ask {A: Ask {A: Done False, B: Done False, C: Done False},
      B: Ask {A: Done False, B: Done False, C: Done True},
      C: Ask {A: Done False, B: Done False, C: Done False}},
  B:
  Ask {A: Ask {A: Done False, B: Done False, C: Done False},
      B: Ask {A: Done False, B: Done False, C: Done False},
      C: Ask {A: Done False, B: Done False, C: Done False}},
  C:
  Ask {A: Ask {A: Done False, B: Done False, C: Done False},
      B: Ask {A: Done False, B: Done False, C: Done False},
      C: Ask {A: Done False, B: Done False, C: Done False}}}

```

(22) Who is (equal to) everyone?

```

> pretty (evalT (lift ask >>= \x ->
                everyoneT >>= \y ->
                return (x == y)))
Ask {A: Done False, B: Done False, C: Done False}

```

We rule out the pair-list reading for the last question if we assume that a quantifier can only take inverse scope over a proposition. (This assumption also helps explain crossover, superiority, and the role of linear order in polarity licensing.)

4. Binding

4.1. Binding among absorbed quantifiers

Chierchia claims to explain binding among quantifiers in the gapped clause.

(23) Who_i did every student_j give every paper_k of his_j to t_i^{j,k}?

$$\begin{aligned} & \text{who}_i + \text{every student}_j, \text{ every paper}_k \text{ of his}_j + t_j \text{ gave } t_k \text{ to } t_i^{j,k} \implies \\ & \lambda P. \exists A. [W(\langle \text{every student}_j, \text{ every paper}_k \text{ of his}_j \rangle, A) \\ & \quad \wedge P(\lambda p. \exists x y f. [A(x, y) \wedge \forall x y. [A(x, y) \implies \text{animate}(f(x, y))]] \\ & \quad \quad \quad \wedge p = \lambda(x \text{ gave } y \text{ to } f(x, y)))]] \end{aligned}$$

where

$$\begin{aligned} & \langle \text{every student}_j, \text{ every paper}_k \text{ of his}_j \rangle \\ & \quad = \lambda A. \text{every student}_j(\lambda x. \text{every paper}_k \text{ of his}_j(\lambda y. A(x, y))) \end{aligned}$$

is a 2-adic quantifier formed out of 2 monadic quantifiers.

This last definition only makes sense if we interpret his_j as captured by λx! So the inputs to Absorption are not generalized quantifiers but syntactic trees or functions from assignments to generalized quantifiers.

4.2. Binding into heads

Chierchia does not address binding connectivity from the gapped clause into the fronted wh-phrase.

(24) Which book_i by her_j is every author_j most proud of?

(25) The book_i by her_j that every author_j is most proud of is her first novel.

Two natural ideas: *book by her* and *every author is most proud of* each denote ...

- a set of functions from individuals to individuals (Jacobson)?
- a function from individuals to sets of individuals (Winter)?

Then, combine the two denotations by intersection (Jacobson) or pointwise intersection (Winter).

But then, we need to rule out apparently unbound pronouns.

(26) a.#Which book_i by her_j is she_j most proud of?

b.#The book_i by her_j that she_j is most proud of is her first novel.

c.#A book_i by her_j is her first novel.