Notes on presupposition and order of composition [1]

Chris Barker, *NYU*
chris.barker@nyu.edu
http://homepages.nyu.edu/~cb125

[This draft was completed on 14 October 2008 in preparation for a discussion in Philippe Schlenker's NYU seminar on presupposition. Today is 14 January 2010, and it's going to be a while before I return to this. Therefore I am making it available in this rough form. Comments, encouragement, criticisms welcome.]

1. Order

It is well known (Karttunen, Heim, etc.) that the presuppositions of complex expressions depend in part on the order of the parts of that expression.

(1) a. France has a king and the King of France is bald.
    b. The King of France is bald and France has a king.

The standard judgment (which I share) is that (1a) asserts rather than presupposes that France has a king. In contrast, (1b) presupposes that France has a king. Assuming that the *and* in (1) is the simple bivalent logical connective, the key difference appears to be the linear order of the conjuncts: in (1b), the information that France has a king arrives too late to help satisfy the presupposition triggered by the definite description in the first conjunct.

In dynamic semantic accounts of presupposition in the tradition of Heim (1983), the linear asymmetry is built into lexical items corresponding to the various logical connectives. But as Soames, Schlenker, and others point out, making these assumptions on a per-lexeme basis missing the systematic pattern that the asymmetry always favors earlier material over later material.

So it is necessary to state the linear asymmetry independently of specific lexical items. But on the standard conception of the division of work between syntax and semantics, semantic composition depends entirely on function argument structure, and not on linear order. It may not even be clear how semantic interpretation *could* be sensitive to linear order. Schlenker concludes that the behavior of presuppositions must crucially be stated in terms of non-semantic objects such as strings or syntactic trees.

[1]Thanks to Doug Bemis, Philippe Schlenker, Chung-chieh Shan, and Anna Szabolcsi.

However, I believe rather that some notion of order not only can, but should be part of semantics (see, e.g., Shan and Barker 2006). On the practical side, theoretical computer science has developed robust techniques for describing languages whose composition is sensitive to order. The usual strategy is to express order sensitivity by mapping an order-sensitive denotation into an order-independent one by means of a continuation passing style transform.

So here is the main issue: on the one hand, we have linear order, a property of strings. On the other hand, we have entailments, a property of meanings. Which side of the line dividing syntax from semantics should statements about presuppositions should be stated? I aim to show that it is perfectly feasible to reconstruct Schlenker's theory of presuppositions within the syntax-semantics interface, without resorting to quantification over strings.

2. Composition

This section motivates the claim that the incremental effects in presupposition projection do not track linear order, but rather order of semantic composition. To put it crudely, it's order at logical form, not linear string order.

Schlenker (2008) [”Be articulate”] suggests that aspects of the behavior of presuppositions in complex expressions can be explained as the interaction of certain quasi-Gricean principles. Like all Gricean maxims, the newly proposed maxims consider what else might have been said, i.e., other possible utterances. In fact, Schlenker's proposed maxim Be Brief provides fairly detailed instructions for constructing alternative utterances:

(2) Be brief:
Given a context set C, a predicative or propositional occurrence of d is infelicitous in a sentence that begins with 'a (d and' if for any expression g of the same type as d and for any sentence completion b, $C \Vdash a(d \text{ and } g)b \Leftrightarrow agb$.

For instance, if a = ”John” and d = ”tried to leave”, then a sentence that begins *John (tried to leave and* will be infelicitous in any context that entails that John tried to leave. (In combination with other assumptions and principles, this allows Schlenker to predict, among other facts, that *John managed to leave* presupposes that John tried to leave.)

There are some unusual elements in (2) compared to most statements of Gricean maxims. For instance, the utterances in question contain parentheses (unlike actual utterances), and there is a notion of the semantic type of an expression. Schlenker provides the following guidance:

(3) It is worth noting that in [(2)], a, b, g and range over *strings of symbols* of a language that includes parentheses to disambiguate structure. Equivalently, one can think of a, b, and g as ranging over *parts of a syntactic tree*, but there is no requirement whatsoever that they may only denote constituents (though of course their values must correspond to continuous strings).

In fact, we shall see that the objects over which Schlenker's syntactically-based theories are stated cannot be strings, but must be fully-disambiguated syntactic trees. (Technically, they will be parts of trees, perhaps as defined in Harrison (1978:382).)

2.1. Disambiguating syntactic structure

Schlenker makes this point in unpublished work using the following type of example:

(4) a. Ann will come and Bob will stay at home with the kids, or they will hire a baby-sitter and Bob will come • too.
b. (A and not B) or B too
c. A and (not B or B too)

Here and below, • marks the point at which we would like to divide the sentence into a left context and a completion. Only the structure in (4c) guarantees that Ann will come, so only the structure in (4c) guarantees that the presupposition of *too* will be satisfied sentence-internally. Therefore the syntactic objects mentioned in (2) must contain enough parentheses to sufficiently disambiguate the syntactic structure.

2.2. Anticipating syntactic structure

The example in (4) involves structure present in the initial half-tree, i.e., material that precedes the presupposition trigger. But a similar point can be made with structure involving material that strictly follows the trigger:

(5) John used to smoke and John • stopped smoking.

Concentrating on the point marked by the bullet (•), we can choose a = *((John (used (to smoke))) (and (John (*. It is easy to see that for any grammatical completion $g)b$, *((John (used (to smoke))) (and (John (g) b* will be equivalent to *((John (used (to smoke))) (and (John ((used (to smoke)) (and g) b*. Schlenker's theory correctly predicts that the initial conjunct in (5) satisfies the presupposition triggered by *stopped*.

(6) John used to smoke and John • stopped smoking last year.

But if we add the temporal phrase *last year*, the initial conjunct is no longer sufficient to entail the presupposition. The reason is that if John stopped smoking last year, that presupposes that he used to smoke at some point last year. The initial conjunct only says that he used to smoke, without entailing that it was sufficiently far in the past to satisfy the presupposition. The solution is to recognize that the initial string a will have a different structure:

(7) a. ((John (used (to smoke))) (and (John (
    b. ((John (used (to smoke))) (and (John ((

The difference is that there will be two parentheses at the end rather than one. This extra paren anticipates that there will be a verb phrase modifier (i.e., *last year*). Because (2) requires us to quantify over all possible syntactic completions, and because at least some of those completions will place temporal restrictions on the second verb phrase, we correctly predict that the initial context of of (6) does not by itself entail the presupposition triggered by *stopped*. In other words, computing presuppositions incrementally requires anticipating whether there will be modifiers further downstream.

The difference between this case and (4) is that in (4), we had to disambiguate the structure of material already in view at the point at which we wanted to compute a presupposition; in (7), we need to distinguish between the structures of material that has not yet been seen. In processing terms, we must assume either that the parser makes a number of hypotheses about what is coming, guessing for instance that the end of the initial segment should have one, two, three, or more parentheses; or else the parser must make its best guess at each point, backtracking if things turn out differently than expected.

These issues do not arise in a dynamic semantics theory of presupposition, since those theories generally begin with a fully parsed syntactic representation in which the syntactic structure reflects full knowledge of material that follows the expression in question. What I am suggesting is that incremental presuppositions do not depend directly on the order in which syntactic expressions are encountered, but rather, on the order in which semantic objects are evaluated (combined).

2.3. Anticipating quantifier scope ambiguity

The next step is to consider a case in which material that follows the evaluation point takes scope over material in the initial context.

(8) a. Someone managed to stop each train.

b. $\forall y \exists x.\mathbf{managed}(\mathbf{stop}\, y\, x)$ Assertion (many alert engineers)
$\forall y \exists x.\mathbf{tried}(\mathbf{stop}\, y\, x)$ Presupposition

c. $\exists x \forall y.\mathbf{managed}(\mathbf{stop}\, y\, x)$ Assertion (one busy hero)
$\exists x \forall y.\mathbf{tried}(\mathbf{stop}\, y\, x)$ Presupposition

The sentence in (8a) has (at least) two readings, depending on whether the quantifier introduced by *each* takes scope over *someone* or not, leading to the two (simplified) readings sketched in (8b) and in (8c). I take it that the presuppositions of the two readings correspond closely to the structure of the scoped readings, as indicated.

In the course of applying Be Brief, we are supposed to think in terms of strings. We would set a = *someone* and d = *tried to stop each train*. We would then need to quantify over all possible ways of completing the bracket-enhanced string *someone (tried to stop each train and*. Among the choices for g will be *managed to stop each train*, and we can choose b empty. We are then faced with the need to assess the following logical equivalence:

(9) Someone (tried to stop each train and managed to stop each train).
$\Leftrightarrow$ Someone (managed to stop each train).

This logical equivalence cannot be evaluated until the quantifier scopings have been fixed. Thus presuppositions must be computed over fully disambiguated logical forms.

To be sure, the need to relativize computation of an implication to a disambiguated meaning is present in standard Gricean inferences as well. For instance, *Someone solved most of the problems* either implicates *no one solved all of the problems* or else *not all of the problems were solved*, depending on how the scope relations in the original sentence are construed. The usual reasoning goes something like this: the speaker could have said such and such, they chose not to, the reason they chose not to may have been respect for Quality, and so on. In this type of reasoning, the original utterance may be compared to a different possible utterance with *all* substituted in for *most*. But only an interpretation of the modified sentence that matches the scope structure of the original will be a viable candidate for consideration as an alternative utterance.

Could we rescue Be Brief through a similar appeal to some notion of viable or salient alternatives? No: rather than comparing the actual utterance with some salient alternative, which is the usual Gricean situation (i.e., existential quantification over salient possible alternative utterances), Be Brief explicitly quantifies over all possible well-formed completions (universal quantification over possible alternative utterances). The reason this innovation is not innocent is that it assumes that every syntactically well-formed completion provides a relevant alternative. But for each completion, only those construals that match the original's quantifier scoping should count (upon pain of concluding that the universal quantification over the logical equivalence is far too hard to satisfy). This suggests that scope relations must be represented in the initial segment, in order to guarantee appropriate semantic uniformity across all possible completions.

But representing scope relations requires anticipating what quantifiers will appear in the completion. Worse yet, different completions will contribute different sets of quantifiers. For instance, in addition to the choice of g in (8) above, we must also consider the following:

(10) John (tried to stop each train and managed to stop each train in each train yard in every small city in each county in each state in every country).

If these quantifiers are construed as taking wide scope over the entire sentence, then their scope must be part of the specification of the initial segment (the segment labelled a in the statement of Be Brief in (2)); but because these extra quantifiers are only introduced within the completion g, and different completions introduce different sets of quantifiers, the initial segment must vary according to the choice of the segment g.

Crucially, in this case, it is not enough to add extra parentheses in the initial segment. In addition, for readings such as (8b), it is necessary to anticipate the quantifier contributed by *each train*. This suggests that presuppositions are not sensitive to linear order of syntactic elements, but rather, to the order in which semantic elements are combined.

The puzzle, from the point of view of a purely syntactic approach to incremental presuppositions, is that disambiguation of quantifier scope relations involves evaluating a quantifier out of its linear sequence.

Or else type shifting in a directly compositional system (Hendriks, Barker and Shan).

[Consider whether it's important to move the quantifier out of the complement of the presupposition trigger.]

2.4. Reconstruction

If presuppositions are indeed calculated on logical form, this predicts that presuppositions can be incrementally satisfied even if the trigger precedes the material that delivers local satisfaction whenever we have independent evidence that the order of semantic composition diverges from linear order.

(11) a. Which of his children did each man call?
b. Which of his children did each father call?

The sentence in (11a) presupposes each relevant man has children. But the variant in (11b) presupposes at most that there are relevant fathers in the domain. (We can embed these sentences under ”know” to avoid complications involving questions.)

In a simple-minded incremental system, the fact that the information that fathers are involved comes linearly after *children* suggests that these are symmetric cases. However, I have a hunch that these are not symmetric cases. Rather, whatever mechanism allows the quantifier to bind the pronoun in the fronted wh-phrase also allows the content (speaking roughly) near the quantifier to satisfy presuppositions in the fronted phrase.

[Could it be symmetric? Arguments?]

2.5. Assessment

Apparently, we must compute presuppositions over structures that are disambiguated syntactically (section 1.1), whose syntactic structures reflect material that follows the presupposition trigger under consideration (section 1.2), and that can represent in the initial segment the semantic contribution of quantifiers that occur after the presupposition trigger. In other words, presuppositions are calculated over fully disambiguated logical forms, not syntactic strings or even (purely) syntactic trees. And since logical forms are (by design) homomorphic to semantic denotations, any generalization over logical form can be rendered as a generalization over semantic objects.

What then might a semantic strategy for incremental presupposition computation look like?

3. Continuations

In computer programming languages, order matters.

(12) a.
```
x := x + 1
x := 1
print x
```

b.
```
x := 1
x := x + 1
print x
```

These two blocks of pseudocode differ only in the order of the first two statements. Nevertheless, the behavior of the two blocks will differ: the first will print the number 1, and the second will print the number 2.

What we need in order to understand this difference is some way of representing the meaning of these blocks that encodes the difference in evaluation order. The danger is that the language that we use to encode the difference will itself be order-sensitive, in which case our analysis is hopelessly circular.

In order to make this problem concrete, consider representing code in the pure lambda calculus. The lambda calculus is confluent, which means that no matter which application you decided to reduce first, you will always arrive at the same result. Could the lambda calculus, then, serve as an order-irrelevant representation of sequential objects?

Well, one problem is that confluence is only guaranteed for those lambda terms that can be fully reduced. If a lambda form contains subterms that do not converge, the order in which we choose to reduce applications matters very much.

In order to illustrate this last point, we need to be explicit about what we mean for a lambda term to be fully reduced. There are three kinds of expressions in the lambda calculus, which I will classify into values and programs:

x	Variable: value
$(\lambda x M)$	Abstract: value
(MN)	Application: program

Here, x schematizes over variables, and M and N schematize over arbitrary lambda terms. In our computational metaphor, we will consider our job done if the term is a value, that is, either a variable or a lambda abstract. But if the term is an application (i.e., a form interpreted as a function M applied to an argument N), we must reduce it if possible, continuing until we have a value. [Because I'll be using Plotkin's (1975) call-by-name CPS transform below, I'm following Plotkin's assumptions here. Many, many other schemes are possible. Also following Plotkin, assume we're only interested in terms that are combinators, i.e., that have no free variables.]

Now we can exhibit a program that cannot be reduced to a value:

$$\Omega = ((\lambda x(xx))(\lambda x(xx)))$$

Because this term is an application (i.e., is of the form (MN) for $M = N = \lambda x.xx$), it is a program, and not fully reduced. If we beta-reduce this form, we get a “reduced” form that is identical to the original form (up to alphabetic variance). The result is itself an application, and therefore not fully reduced. This is the simplest “infinite loop” in the pure lambda calculus.

Now we can examine a term where order of reduction makes a big difference. Let **I** be the identity function (λxx). Then

$$((\lambda x\mathbf{I})\Omega) = ((\lambda x(\lambda xx))\,((\lambda x(xx))\,(\lambda x(xx))))$$

This is an application, so we need to reduce. However, we have two choices for what to reduce first: we can reduce the leftmost application first, resulting in (λxx), a value. Success! Or we could start by reducing the rightmost application, in which case we’ll begin an endless series of profitless reductions, never getting closer to a value.

Obviously, in this case, we want to start on the left. As long as we agree to always reduce the leftmost application first, we can be sure that we will always arrive at a value (that is, for any expression that reduces to a value).

But we’d like to make this policy of always working from left to right explicit and precise, perhaps by encoding it in the form of a program. And since the pure lambda calculus is Turing complete, we might naturally choose to encode the reduction algorithm in the pure lambda calculus. But if we do this, then the reduction algorithm will be expressed in a language whose meaning depends on evaluation order, and we’re right back where we started.

3.1. Continuations to the rescue

Continuations have many uses: enhancing programming language expressivity, compiler optimization, characterizing control constructions, revealing logical symmetries, and more. For present purposes, what is most important is that they are especially useful for reasoning about the order in which computations unfold.

Continuations provide a way to break the meta-circularity. What we are going to do is show how to encode the original lambda term in a form that implements the desired evaluation strategy in a way that does not itself depend on which evaluation strategy is being used.

Plotkin (1975:153) recursively defines what is know as a call-by-name (CBN) continuation passing style (CPS) transform $[\bullet]$ as follows:

(91)
$$\begin{aligned}[x] &= x\\ [\lambda x M] &= \lambda\kappa.\kappa(\lambda x[M])\\ [MN] &= \lambda\kappa.[M](\lambda m.m[N]\kappa)\end{aligned}$$

Thus [•] maps lambda terms into (more complex) lambda terms. For instance, the mapping rules apply to our problematic term as follows:

$$\begin{aligned}[(\lambda x\mathbf{I})\Omega] &= \lambda\kappa.[\lambda x\mathbf{I}](\lambda m.m[\Omega]\kappa)\\ &= \lambda\kappa.(\lambda\kappa.\kappa(\lambda x[\mathbf{I}]))(\lambda m.m[\Omega]\kappa)\end{aligned}$$

We shall see that this transformed term encodes the computation expressed by the original term in such a way that the the desired order of evaluation is explicitly encoded. Since the transform is an abstract (it begins with "$\lambda\kappa$"), it's already a value, so we can't evaluate the transformed expression directly. In order to see the encoding unfold, we must apply this term to the trivial continuation, **I**, which triggers the reduction process. As reduction proceeds, you will see that unlike the original term, at each stage there is exactly one possible reduction, so we can't possibly make the wrong choice:

$$\begin{aligned}([(\lambda x\mathbf{I})\Omega]\mathbf{I}) &= ((\lambda\kappa.(\lambda\kappa.\kappa(\lambda x[\mathbf{I}]))(\lambda m.m[\Omega]\kappa))\mathbf{I})\\ &= ((\lambda\kappa.\kappa(\lambda x[\mathbf{I}]))(\lambda m.m[\Omega]\mathbf{I}))\\ &= ((\lambda m.m[\Omega]\mathbf{I})(\lambda x[\mathbf{I}]))\\ &= (((\lambda x[\mathbf{I}])[\Omega])\mathbf{I})\\ &= (\mathbf{I}\,\mathbf{I})\\ &= \mathbf{I}\end{aligned}$$

As the zipper descends, there is at most one application that is not hidden underneath an abstract (recall that since abstracts are already values, we don't perform reductions inside of an abstract). As a result, at each step there is only one possible move. The CBN CPS transform forces us to always reduce the leftmost application first, ignoring the inner structure of the argument Ω.

Incidentally, the reason it makes sense to call this transform "call-by-name" is that the argument (in this case, the argument is Ω) is passed to the functor unevaluated. In linguistic terms, it corresponds to a de dicto interpretation: if Lars wants to marry a Norwegian, then the description "a

Norwegian" is incorporated into the description of his desire unevaluated, rather than first picking out a specific de re individual (i.e., by evaluating the direct object, call-by-value) and then supplying the individual to build the desire representation.

If we had used Plotkin's call-by-value transform, we would have been forced to try to reduce the argument first, leading us into the infinite loop. For those readers who would like to see this for themselves, here is Plotkin's call-by-value transform:

$$\begin{aligned} [x] &= \lambda\kappa.\kappa x \\ [\lambda x M] &= \lambda\kappa.\kappa(\lambda x[M]) \\ [MN] &= \lambda\kappa.[M](\lambda m.[N](\lambda n.mn\kappa)) \end{aligned}$$

What will be used below: not infinite loops; application to trivial continuation. Usually side effects: binding, quantification, licensing; in the discussion below, presupposition licensing, which a peculiar mixture of content (entailments) and licensing.

4. Incremental composition

For the incremental computation of semantic commitments, we need something that I will provisionally call a half-continuation, i.e., a system that keeps track of left context only.

[Computationally: mutable state. Linguistically, de Groote's reconstruction of DPL. Difference between that and here: we track incremental computation of content, not just discourse referents.]

Notational conventions: $\lambda xy.\phi = (\lambda x(\lambda y\phi))$, and application is left-associative, so that $xyz = ((xy)z)$.

We will arrive at a system for computing incremental presuppositions in three stages: building a simple system that has standard function/argument combination (i.e., function application); continuizing lexical items; and continuizing function/argument combination.

To start, at the most basic level (i.e., before we continuize), to make function application sufficiently explicit, we will need two combinators: one for forward combination (**I** = $\lambda xy.xy$) and one for backward combination (**T** $= \lambda xy.yx$):

$$
\begin{aligned}
&(\mathbf{T}[\text{John}](\mathbf{T}(\mathbf{I}[\text{realized}][\text{it}])[\text{yesterday}]))\\
=\ &(\mathbf{T}[\text{John}](\mathbf{T}((\lambda xy.xy)[\text{realized}][\text{it}])[\text{yesterday}]))\\
=\ &(\mathbf{T}[\text{John}](\mathbf{T}([\text{realized}][\textbf{it}])[\text{yesterday}]))\\
=\ &(\mathbf{T}[\text{John}]((\lambda xy.yx)([\text{realized}][\textbf{it}])[\text{yesterday}]))\\
=\ &(\mathbf{T}[\text{John}]([\text{yesterday}]([\text{realized}][\text{it}]))\\
=\ &((\lambda xy.yx)[\text{John}]([\text{yesterday}]([\text{realized}][\text{it}]))\\
=\ &(([\text{yesterday}]([\text{realized}][\text{it}]))[\text{John}])
\end{aligned}
$$

In, for example, in Heim and Kratzer (1998), it is not necessary to explicitly specify which of the two modes of combination are relevant, since it is possible to decide which element is the functor and which is the argument by examining the semantic types. We could do that here, but it will be useful to be more explicit.

Second stage: Continuizing at the lexical level: if a lexical item has denotation x, then its continuized meaning will simply be the LIFTed version of x, that is, $\mathbf{T}x = \lambda\kappa.\kappa x$. For instance, the continuized version of the denotation of *John* is $\lambda\kappa.\kappa\mathbf{j}$, the continuize version of the denotation of *left* is $\lambda\kappa.\kappa\mathbf{left}$, and so on.

Third stage: in order to continuize the system, we need an operator **H** (for 'Half-continuation') that applies to each of **I** and **T**:

$$\mathbf{H} = \lambda MLR.\lambda\kappa.R(L(\lambda lr.\kappa(Mlr)))$$

So that

$$\mathbf{HI} = \lambda LR.\lambda\kappa.R(L(\lambda lr.\kappa(lr)))$$

and

$$\mathbf{HT} = \lambda LR.\lambda\kappa.R(L(\lambda lr.\kappa(rl)))$$

We continuize at the level of syntactic composition by replacing plain functional application (either **T** or **I**) with their continuized versions (**HT** and **HI**, respectively), and by replacing lexical items with their continuized versions.

So for

John ((realized it) yesterday)

we have

$$
\begin{aligned}
&\mathbf{HT}\ [\text{John}]\ (\mathbf{HT}\ [\text{realized it}]\ [\text{yesterday}])\\
=\ &\lambda\kappa.(\mathbf{HT}[realized\ it][yesterday])([John](\lambda lr.\kappa(rl)))\\
=\ &\lambda\kappa.(\mathbf{HT}[realized\ it][yesterday])([\lambda\kappa.\kappa\mathbf{j}](\lambda lr.k(rl)))\\
=\ &\lambda\kappa.(\mathbf{HT}[realized\ it][yesterday])(\lambda r.\kappa(r\mathbf{j}))\\
=\ &\lambda\kappa.(\lambda\kappa.[yesterday]([realized\ it](\lambda lr.\kappa(rl))))(\lambda r.\kappa(r\mathbf{j}))\\
=\ &\lambda\kappa.[yesterday]([realized\ it](\lambda lr.(\lambda r.\kappa(r\mathbf{j}))(rl)))\\
=\ &\lambda\kappa.[yesterday]([realized\ it](\lambda lr.\kappa(rl\mathbf{j})))\\
=\ &\lambda\kappa.(\lambda\kappa.\kappa(\mathbf{yesterday}))([realized\ it](\lambda lr.\kappa(rl\mathbf{j})))\\
=\ &\lambda\kappa.[realized\ it](\lambda lr.\kappa(rl\mathbf{j}))\mathbf{yesterday}
\end{aligned}
$$

As usual with continuations, we finish a derivation by applying the result to the trivial continuation (the identity function, $\lambda x.x$). Doing so gives

$$[realized\ it](\lambda lr.rl\mathbf{j})\mathbf{yesterday}$$

where **j** is John.

The system is designed in such a way that all of the semantic content provided by the left context of an expression is packaged into that expression's semantic argument, its half-continuation. In the example at hand, the half-continuation of *realized it* is the content of the semantic argument to the (continuized) denotation of the phrase, namely,

$$\lambda lr.rl\mathbf{j}$$

Once the values of the semantic arguments l and r are provided (by the completion of the sentence), this continuation will guarantee that John (**j**) did something (l) in some manner (r), i.e., that $(rl)\mathbf{j}$ will hold for some as yet unspecified choice of r and l.

It will help to immediately consider an example in which the left context is non-trivial.

(13) It rained, and John realized it.

Let $\wedge$ be the (curry'd) ordinary propositional conjunction from predicate calculus, bivalent and completely ignorant of any sort of linear asymmetry whatsoever. Since *and* is a lexical item, of course, its continuized version will be $\lambda\kappa.\kappa\wedge$.

It rained and John realized it:
= $\mathbf{HT}[\text{it rained}](\mathbf{HI}[\text{and}](\mathbf{HT}[\text{John}][\text{realized it}]))$
= $\lambda\kappa.(\mathbf{HI}[\text{and}](\mathbf{HT}[\text{John}][\text{realized it}]))([\text{it rained}](\lambda lr.\kappa(rl)))$
= $\lambda\kappa.(\mathbf{HI}[\text{and}](\mathbf{HT}[\text{John}][\text{realized it}]))((\lambda\kappa.\kappa\mathbf{rain})(\lambda lr.\kappa(rl)))$
= $\lambda\kappa.(\mathbf{HI}[\text{and}](\mathbf{HT}[\text{John}][\text{realized it}]))(\lambda r.\kappa(r\mathbf{rain}))$
= $\lambda\kappa.(\lambda\kappa.(\mathbf{HT}[\text{John}][\text{realized it}])([\text{and}](\lambda lr.\kappa(lr))))(\lambda r.\kappa(r\mathbf{rain}))$
= $\lambda\kappa.(\lambda\kappa.(\mathbf{HT}[\text{John}][\text{realized it}])((\lambda\kappa.\kappa\wedge)(\lambda lr.\kappa(lr))))(\lambda r.\kappa(r\mathbf{rain}))$
= $\lambda\kappa.(\lambda\kappa.(\mathbf{HT}[\text{John}][\text{realized it}])(\lambda r.\kappa(\wedge r)))(\lambda r.\kappa(r\mathbf{rain}))$
= $\lambda\kappa.(\mathbf{HT}[\text{John}][\text{realized it}])(\lambda r.(\lambda r.\kappa(r\mathbf{rain}))(\wedge r))$
= $\lambda\kappa.(\mathbf{HT}[\text{John}][\text{realized it}])(\lambda r.\kappa(\wedge r\mathbf{rain}))$
= $\lambda\kappa.(\lambda\kappa.[\text{realized it}]([\text{John}](\lambda lr.\kappa(rl))))(\lambda r.\kappa(\wedge r\mathbf{rain}))$
= $\lambda\kappa.(\lambda\kappa.[\text{realized it}]((\lambda\kappa.\kappa\mathbf{j})(\lambda lr.\kappa(rl))))(\lambda r.\kappa(\wedge r\mathbf{rain}))$
= $\lambda\kappa.(\lambda\kappa.[\text{realized it}](\lambda r.\kappa(r\mathbf{j})))(\lambda r.\kappa(\wedge r\mathbf{rain}))$
= $\lambda\kappa.[\text{realized it}](\lambda r.(\lambda r.\kappa(\wedge r\mathbf{rain}))(r\mathbf{j}))$
= $\lambda\kappa.[\text{realized it}](\lambda r.\kappa(\wedge(r\mathbf{j})\mathbf{rain}))$

Finishing by applying the result to the trivial continuation, we see that the half-continuation (the semantic argument) for *realize it* is

$$\lambda r.\wedge(r\mathbf{j})\mathbf{rain}$$

Once the semantic argument (r), corresponding to the verb phrase, gets instantiated by the completion, this continuation will guarantee that John does r, and that it is raining. Put another way, no matter what value r takes on, we can still be sure *from the left context alone* that the sentence will entail that it is raining.

The next example to try is one in which the order of the conjuncts is reversed:

John realized it and it rained.

(The intention here is that ”realized it” is interpreted as ”realized it rained”.)

$\textbf{HT}(\textbf{HT}[\text{John}][\text{realized it}])[\text{and-it-rained}]$
$= \lambda\kappa.[\text{and-it-rained}](\textbf{HT}[\text{John}][\text{realized it}])(\lambda lr.\kappa(rl)))$
$= \lambda\kappa.[\text{and-it-rained}]((\lambda\kappa.[\text{realized it}]([\text{John}](\lambda lr.\kappa(rl))))(\lambda lr.\kappa(rl)))$
$= \lambda\kappa.[\text{and-it-rained}]((\lambda\kappa.[\text{realized it}]([\lambda\kappa.\kappa\textbf{j}](\lambda lr.\kappa(rl))))(\lambda lr.\kappa(rl)))$
$= \lambda\kappa.[\text{and-it-rained}]((\lambda\kappa.[\text{realized it}](\lambda r.\kappa(rj)))(\lambda lr.\kappa(rl)))$
$= \lambda\kappa.[\text{and-it-rained}]([\text{realized it}](\lambda r.(\lambda lr.\kappa(rl))(r\textbf{j})))$
$= \lambda\kappa.[\text{and-it-rained}]([\text{realized it}](\lambda r.(\lambda ls.\kappa(sl))(r\textbf{j})))$
$= \lambda\kappa.[\text{and-it-rained}]([\text{realized it}](\lambda r.(\lambda s.\kappa(s(r\textbf{j})))))$
$= \lambda\kappa.[\text{and-it-rained}]([\text{realized it}](\lambda rs.\kappa(s(r\textbf{j}))))$

After application to the trivial continuation, we get

$$\lambda rs.s(r\textbf{j})$$

as the half-continuation for *realized it*. This says that at the moment immediately before evaluating *realized it*, all we know is that John (**j**) is going to do something (r), and the proposition ($r\textbf{j}$) is going to serve as the argument to some function s. After we hear the rest of the sentence, we come to realize that s will be the function $\lambda p. \wedge \textbf{rain}p$, but we don't know that right before *realize it* is going to be evaluated.

Some other abbreviated examples:

(14) a. It didn't rain or John • realized it. $\lambda l. \vee (l\textbf{j})(\neg\textbf{rain})$
b. If it rained, John • realized it. $\lambda l.\textbf{rain} \rightarrow (l\textbf{j})$

What we know is that if it's raining, John must have some property (l). This is logically equivalent to the claim that if it's raining, John realizes it.

4.1. Reconstructing Schlenker's theory of presupposition projection

With half-continuations in our toolbox, we can turn to reconstructing Schlenker's theory of incremental presupposition projection.

Adopting Schlenker's foundational assumptions about context (roughly, a Stalnakerian view on which contexts are modeled as sets of worlds), and adopting his notation, assume that we are evaluating a sentence s with respect to a context set C. The sentence s contains within it an expression

e that contributes content d' to the assertive component of the meaning of s. In addition, e triggers the presupposition d. We wish to predict whether the context set, in combination with the semantic information contributed by the material in s that linearly precedes e, satisfies the presupposition d.

Assume, then, that $s = aeb$. Here, a and b are half-trees (for Schlenker, strings with potentially unbalanced parentheses). We suppose that c' is the strongest expression (strongest in the sense of entailments) such that $C \Vdash ag'b' \Leftrightarrow a(c' \wedge g')b'$ for any choice of g' and b' (as long as $a(c' \wedge g')b'$ is a grammatical sentence). Given an expression e that asserts d' and presupposes d, the presuppositions of e will be incrementally satisfied by a half-tree a just in case $C \Vdash c' \rightarrow d$.

For instance, applying this definition to the example *It rained, and John realized it* (discussed above), we have a = *((it rained) (and (John (*. Following the reasoning in the Local Contexts paper, we see that the strongest c' is (some syntactic expression whose meaning is the same as) *it rained*. Since c' does indeed entail the presupposition of *realized it rained*, we conclude that the presupposition of e are incrementally satisfied relative to this initial half-tree.

We can easily reconstruct this notion in the continuation analysis. An expression e asserting d' and presupposing d will have its presupposition incrementally satisfied just in case $C \wedge \kappa = C \wedge \lambda g.\kappa(d \wedge g)$, where κ is the half-continuation for the expression whose local context we are computing. This says that adding the presupposition does not affect the semantic outcome (given a context set C). [Note on conjoining a context set with a proposition.]

We have e = *realized it*, and e's presupposition $d = \lambda x.\textbf{rain}$. The half-continuation κ of e in *((it rained) (and (John e)))* as calculated above under (13) is $\kappa = \lambda l.(l\textbf{j}) \wedge \textbf{rain}$ (with $\wedge$ written as an infix for readability). Then

$$\begin{aligned}\lambda g.\kappa(\textbf{rain} \wedge g) &= \lambda g.(\lambda l.(l\textbf{j}) \wedge \textbf{rain})(\lambda x.\textbf{rain} \wedge g)\\ &= \lambda g.((\lambda x.\textbf{rain} \wedge g)\textbf{j}) \wedge \textbf{rain}\end{aligned}$$

Now we observe that

$$\lambda l.(l\textbf{j}) \wedge \textbf{rain} = \lambda g.((\lambda x.\textbf{rain} \wedge g)\,\textbf{j}) \wedge \textbf{rain}$$

It follows that the presupposition of e will be incrementally satisfied no matter what the context set will turn out to be.

Note that in contrast to the syntactic formulation, there is no quantifying over possible completions. This is not a deep difference, since two functions to be equivalent just in case they give the same result for every argument.

However, there is an important difference in what these two formulations say about the deeper nature of presupposition. The syntactic formulation quantifies over possible syntactic completions, i.e., over strings that grammatically complete the initial segment. In the semantic formulation, the quantification is over meanings of the appropriate type to serve as arguments to the half-continuation.

Which formulation is the right one? It appears that the syntactic properties of the completions mentioned in the syntactic formulation are entirely ignored, and they are needed solely for their semantic contribution.

If we quantify over syntactic objects, we would naturally expect that the crucial test would be expressed in terms of syntactic well-formedness, for example, "... e has its presupposition incrementally satisfied just in case for every expression g' and completion b', $ag'b'$ is grammatical just in case $a(dandg')b'$ is." Instead, there are semantic constraints on the choice of expressions (for instance, g must have the same semantic type as e), and the crucial test is expressed in terms of semantic equivalence.

The status of c' in the syntactic formulation has a different status. This c' is needed purely for the sake of its denotation x. The requirement that x be the strongest semantic object satisfying the equivalence forced c' to express exactly the content contributed by the initial half-tree a. In the semantic formulation, the contribution of the initial half-tree is computed directly. There is no need to guess what the correct meaning is, since the compositional semantics delivers it automatically. To the extent that the local contexts story is motivated by processing considerations, the fact that the semantic formulation exhibits an algorithm for computing the meaning required by the processing story distinguishes it from the syntactic approach.

What I am suggesting is that if the key requirement involves comparing meanings, the requirement should naturally be expressed in terms of meanings in the first place. Nor is the fact that it can be stated over syntactic strings particularly interesting, given that we are dealing with disambiguated syntactic trees. The fact that semantic generalizations can be stated in terms of the syntactic objects that denote them follows from the usual Montagovian assumptions that the meaning relation is a homomorphism.

To-do list:

- Define incremental presupposition
- Mention Mcnally/Beaver objection as evidence that we're dealing in meanings.

- Demonstrate that the *manage to stop every train* example can be handled. Argument for direct compositionality?
- Generalize algorithm to any compositional system
- Deny that saying ‘at some level of analysis...’ is necessary.
- Comment on Rothschild: many embedings.
- Full continuations for symmetric readings and for crossing satifaction.