Phase Stitching

Generative Grammatik des Südens
Universität Leipzig, October 22nd, 2016

Thomas McFadden
Zentrum für allgemeine Sprachwissenschaft (ZAS), Berlin
mcfadden@zas.gwz-berlin.de

1 Standard phase theory and the problem with Transfer

One of the central observations that has driven syntactic theory for decades is locality, attenuated by successive-cyclicity:

(1) **Locality**
Syntactic operations and relationships are fundamentally required to apply within a local domain.

(2) **Successive-cyclicity**
However, local relationships can be chained together under certain restricted circumstances, deriving what on the surface appear to be non-local relationships.

The standard implementation of these ideas in current Minimalism is in terms of **phases** (Chomsky 2000, 2001, etc.). Phases achieve locality with successive-cyclicity by being **opaque domains** with **transparent edges**:

- What's inside one phase is essentially invisible to what's inside another. They are **encapsulated** from one another and thus cannot interact...
- ...except that the upper part — the edge — of a given phase is transparent, i.e. visible to the phase above. Edges thus function as **escape hatches**, allowing us to derive **successive-cyclicity**.

The basic idea of opaque domains with escape hatches has been around for a while and comes in many versions. The innovation of phase theory is in how it is connected to parts of the derivation:

☞ The big idea is that there's an operation Spell-out or Transfer which periodically submits completed chunks of structure to the interfaces.

☞ Chunks of structure that have been Transferred cannot interact with later steps of the derivation, creating opacity effects — the **Phase Impenetrability Condition (PIC)**.

☞ Crucially, what is Transferred is not the complete structure created up to that point — that would yield absolute locality, i.e. no interactions across phases at all.

☞ Rather, the top bit of the current structure sticks around to participate in the next phase of the derivation, deriving successive-cyclicity through this escape hatch.

A bit more concretely:

- Assume that a head H defines a phase P. Upon completion of P, the complement of H, which we call the **Domain** of P, is sent to Spell-out.
- The **Edge** of P, consisting of the head H and any specifiers and adjuncts, remains, and will only be sent to Spell-out as part of the domain of the next phase up.

This straightforwardly gives us locality with successive-cyclicity.

☞ From here on out I will refer to this standard version of phase theory in terms of its defining feature as **(Phase) Transfer**.

I would like to argue that there is something problematic about Transfer, though it's a bit hard to nail it down because of a lack of precision — and I think a bit of confusion — in the details of its formulation. Here's the question:

? What exactly does Transfer do, and how does it relate to the PIC?

I think there are two broad kinds of answers, both of which seem to be entertained or assumed by different researchers without a whole lot of discussion:

Transformative Transfer: Transfer modifies the phase domain that it operates on or even removes it from the syntactic structure in order to feed it to the interfaces.

Interpretive Transfer: Transfer takes a snapshot of the phase domain to feed into the interfaces, but does not substantially alter its narrow syntactic form.

☞ Each of these options has something to speak for it, but in the end they both have their problems.

The advantage of Transformative Transfer is that it straightforwardly derives the PIC:

- If we literally remove the phase domain from the narrow syntax and ship it off to the interfaces, it simply won't be around to participate in subsequent steps of the derivation, yielding opacity.
- Similarly, if we assume along the lines of Uriagereka (1999) that Spell-out converts the domain from a hierarchical structure to a linearized string, even though still present it will be illegible to syntax.

The problem is that, because it relies on destroying or hiding structural relations, it leads to issues when we try to interpret the complete structure at the end of a derivation:

- If phase domains are actually removed from the structure, they must be reassembled or reassociated with each other in the right order at the interfaces to yield something that can be pronounced and interpreted as a unified sentence or utterance.
- If the domains are retained, but have their structure modified so as to make them syntactically opaque, we need a mechanism that can at least partially recover some structure sensitivity later in order to capture interpretive effects that seem to involve (large parts of) the entire structure, e.g. certain types of reconstruction.

- I will refer to this as the **reconstitution problem**, since it revolves around reconstituting structure that has been destroyed or modified by transformative Transfer.

The advantage of Interpretive Transfer is essentially that it doesn't suffer from the reconstitution problem:

- The phase domain remains in the structure, with its syntactic properties presumably unmodified (though with some indication that information about it has been Transferred to the interpretive components).
- At the end of the derivation it is thus exactly where it needs to be to play its role in the completed structure and potentially interact with late interpretive processes like reconstruction.
- Chomsky (2008), though not entirely explicit, seems to be assuming some version of this, since he wants to leave open the possibility that Agree can see into a complete phase domain, though apparently Merge cannot.

The problem for Interpretive Transfer is that it fails to clearly relate the PIC to Transfer:

☞ Precisely because Transfer doesn't actually modify the syntactic structure of phase domains it applies to, it cannot be used to derive the effects of the PIC.

☞ Contrary to what at least I had always thought, Chomsky (2008) actually just stipulates the PIC.

☞ And this is particularly worrisome, because it's clear that Transfer and the PIC are supposed to apply to the same chunks of structure, yet it comes out as a coincidence.

I would like to suggest that the source of all of these problems is the way that Phase Transfer goes about deriving the attenuated opacity effects of successive-cyclic locality:

- Whether we adopt a version of Transformative Transfer or Interpretive Transfer, the unifying idea is that something has to happen periodically to create opacity and enforce locality.
- The central path of the derivation builds up a single unified structure, and left to its own devices this would lead to everything in the structure being able to interact with everything else.
- In order to break up the derivation into manageable chunks and to enforce locality, the periodic intervention of Transfer and the PIC is assumed.

But here's the thing:

☞ If it is to actually derive the PIC, Transfer has to be given teeth. It has to actually create a disconnect in the structure to prevent non-local things from interacting. And this screws up the unified structure we want in the end.

☞ If we want a unified structure, Transfer has to be non-invasive, but then we don't actually get the PIC. We have to simply stipulate it on top to get locality.

The lesson I would like to take from this is the following:

☞ The unified structure for the complete sentence or utterance is the wrong level to try to impose locality constraints, precisely because it's where everything gets together in one place.

☞ Of course, we need to have the unified structure at the end of the derivation so that it can be deployed by the interfaces for pronunciation and interpretation.

☞ So locality must actually belong somewhere earlier in the derivation, **before** everything has been put together.

2 On modularity

The way that I want to get this to work is suggested by an observation about the job that phases are supposed to do and the way that they work.

(3) As opaque domains with transparent edges, phases are highly reminiscent of the basic design of a **modular** system.

There are different kinds of modular systems, but modular design is built around having a series of objects that operate for the most part independently, each doing its own job.

☞ This is facilitated by encapsulating each object, hiding its internal information from the other objects so that each can be defined and operate without the knowing about the internal affairs of the others.

☞ This can simplify the design and use of the system and avoid problems where two objects accidentally get in each other's way by messing with each other's stuff.

☞ But since the objects ultimately need to cooperate to build a complex system, they do need a way to communicate with each other, at least a little bit.

☞ The way to do this while maintaining the greatest degree of encapsulation is if the system provides well-defined and restricted interfaces between the objects, and all communication has to proceed via these interfaces.

Just as an aside, as far as I'm aware there are two broad ways for this kind of modularity to arise in a system:

1. It can be a natural and necessary consequence of the different parts of a system operating on completely different kinds of information, which we might think of as being represented with different alphabets.
    - One module cannot interfere with the information proper to another because it literally is not equipped to understand or manipulate it.
    - An example would be the different modules corresponding to linguistic levels, like semantics, syntax, phonology and phonetics. The module for semantic interpretation can't screw things up for the one for phonological computation because it has no idea what to with syllables or features like [+ATR].

- The interfaces between such modules are often transducers, whose job is to literally translate or map information in one alphabet to one in another, like going from phonological features to articulatory phonetic gestures (see e.g. Hale 2007).

2. It can be a design choice, not directly forced in specific places as in the first type, but motivated by the idea that simplicity and reusability of the parts lead to simplicity and robustness of the whole.

   - Individual modules may not be distinguished by the kinds of information that they operate on, but the overall task is made easier by breaking it up into discrete parts that can be easily combined with each other and reused to accomplished complex tasks.
   - An example would be the way that most computer programs are built up out of pieces like subroutines, functions, objects and packages, all of which use the same alphabet, but are able to do their jobs and combine with each other flexibly precisely because they leave each other alone internally.
   - The interfaces between these kinds of modules don't necessarily need to translate between different types of information. Instead they establish how they fit together and what each expects from any others.

To see how this works, consider a very simple example from a program that can do some basic geometric calculations.

- Let's say we a module `compute_circumference`, which takes a number representing the radius of a circle, and returns its circumference.

- To keep things simple, we can have this module really do just that calculation. It would then team up with other modules to interact with the outside world.

- So we can have modules like `compute_area`, `get_from_user` that queries the user for numerical input, `get_from_disk` that retrieves information from storage, `print` that outputs results to the user's screen, and `store_to_disk` that for saving things.

- Each of these modules will be defined to have an interface, laying out what kind of information it takes in (e.g. a number for `compute_circumference`, a string for `print`, a chunk of information and a label or filename to identify a location for `store_to_disk`), what kind of information it spits out (a number for `compute_area`, perhaps a message about success or failure for `store_to_disk`), and what kind of external resources it needs to interact with (e.g. the harddrive or a terminal).

- These interfaces then determine the different ways we can combine the modules together to get things done, e.g. using `get_from_user` to get the radius of a circle, which is fed to `compute_circumference` to figure out its circumference, which is fed to `store_to_disk` to save the result.

Here's where the modular stuff pays off:

☞ Reading from and writing to disk, getting user input and printing output are things we'll want to do in combination with any of the different types of calculation — area of a circle, volume of a cube, hypotenuse of a right triangle.

☞ By separating each out as a module, we only have to write the code for it once and let it be reused by all of these other functions rather than redoing the work of e.g. `store_to_disk` once for each calculation.

☞ By preventing the modules from seeing each other's code, beyond the bare necessity of the interfaces, we avoid unexpected interactions, e.g. several modules can operate internally with variables with reasonable names like `area` or `x` without getting confused with each other.

☞ And we can make changes internal to a module (like if we learn about an exciting new algorithm for computing the area of a square or want to add extra precision to the value we use for π) and know that we won't screw up its interactions with the others, as long as we keep the interfaces stable.

Clearly, to the extent that phases can be thought of as modules, they are modules of this latter type:

- The information they deal with — syntactic features and structures — is all from the same alphabet, so splitting them up has to be motivated by considerations of efficient design.
- Indeed, the idea is supposed to be at least in part that doing derivations phase-by-phase will simplify computation, among other things by enforcing locality.

3 Phase Stitching

Now, my contention is that if we think about phases seriously from this modular perspective, we arrive at a rather different way of doing things than Phase Transfer, which will allow us to derive the PIC without running into the reconstitution problem.

☞ The key ideas here are that modularity is all about reuse and recombination of individual modules, and that the interfaces tell you about how modules can be combined with each other.

☞ If phases are modules, then we should be thinking about them not in terms of taking the unified structure created by a derivation and pulling it apart with Transfer.

☞ Rather, we should be thinking about putting phases together to create the final unified structure.

Here's the quick version:

- Assume that phases are created independently, each built up and manipulated in its own separate **Workspace**. Only when they're completed can they be **Stitched** together to yield the final complete structure.
- All locality-sensitive operations have to take place on the individual phases in their Workspaces before they are stitched together.

- Transparency at the edges is achieved by having the edges be literally shared by two adjacent phases, which allows them to communicate, and ensures above all that only the right kinds of phases can be stitched together.

I'm going to call this approach **(Phase) Stitching**. Here's how it's going to overcome the issues we identified for Transfer:

☞ Familiar syntactic operations like Merge and Agree take place within Workspaces, at an early stage of the derivation before the various phases have been Stitched together.

☞ The later stages of the derivation are all about Stitching the individual phases together to create a complete structure, with no modfication of that structure or loss of information.

☞ This means that we get a nice unified structure at the end that can be deployed for production and interpretation with no need for reconstitution.

☞ Still we derive the PIC from how phases work by having everything that is sensitive to locality happen **before** the different phases get put together, so they literally can't see each other.

4 Some details of the implementation

First some important assumptions about Workspaces:

- Each phase is built up in its own Workspace. The Workspaces are completely encapsulated from each other such that their internal workings cannot take each other into account in any way.
- More or less standard versions of Agree and Merge (including internal Merge) apply within the Workspace and only within the Workspace, thus they cannot involve material from two different phases.
- Each Workspace contains a single uniquely rooted structure. This means that there is no sideward movement, but also that complex left branches (internally complex specifiers and adjuncts) must be created by internal Merge or come from another Workspace, i.e. they must be phases.

This pretty straightforwardly gets us the opacity part of PIC:

☞ Dependencies involving Merge or Agree between elements in two different phases will be unstatable, since these operations apply within a single Workspace.

The way that we get our phases to combine together and interact in the limited ways observed under successive-cyclicity is, like any good modules, via their interfaces.

- In the case of phases, the interfaces are the Edges, whose job is to lay out the protocol for how phases can combine and communicate with each other.

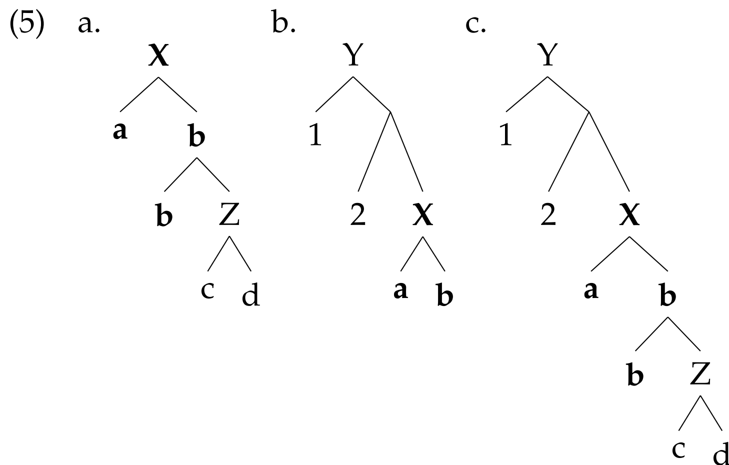
- The mechanism for this combination and communication is the operation Stitch, which essentially unifies matching Edges, thus bringing phases together.
- By hypothesis, Stitch cannot have any effect on the phases it stitches together (something like the No Tampering Condition), but can only make sure that they fit together properly and create the larger structure out of them.

Here's how it works:

(4) **Stitch with edge-matching**
Stitch applies to two phases X and Y, where the Edge at the root of X Matches an Edge interior to Y, yielding an output in which those Edges become a single unified structure. The Edge of a phase is its defining head along with any specifiers and adjuncts.

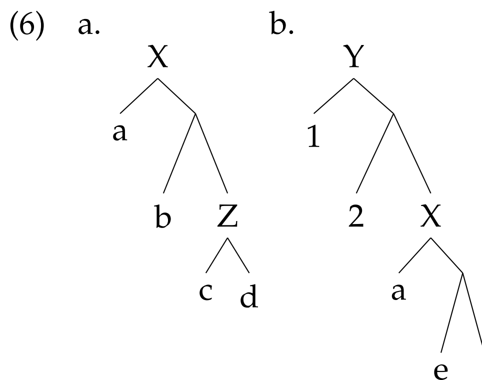
- So we can Stitch 5a with 5b to yield 5c, because the outer edge of 5a (everything above Z) Matches with an inner edge of 5b (everything from X down):

(5)



However, there's no way to Stitch together the structures in 6a and 6b, because they don't have any matching edges:

(6)



Note note what Stitch does for us:

1. It gives us a simple way to describe dependencies between phases.
    - So we can set things up with the specification of the Edges so that a C phase will combine with a *v* phase.

- And we can do more interesting things to make sure that a *v*P phase built on *try* will only Stitch together with a phase corresponding to a control infinitive.

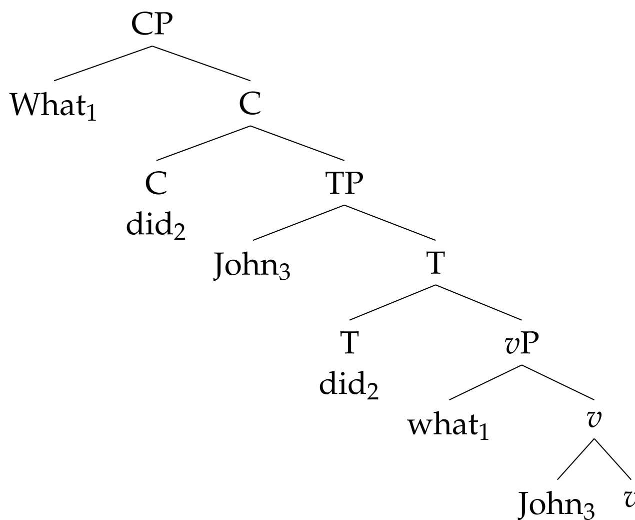
2. It gives us an escape hatch between phases.

   - In order for Stitch to work, the two phases getting together will **both** have to contain a copy of the Edge that they share.
   - This means that something in the Edge will be able to participate in local relationships in both phases, which come to be unified with each other when the two phases Stitch on that Edge.

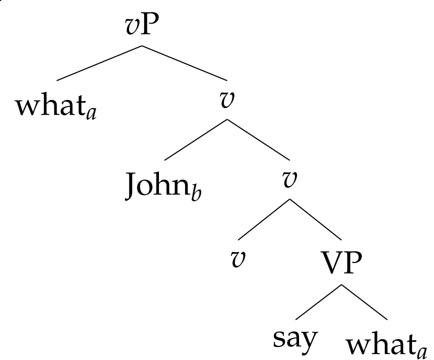
As a quick demonstration, here are the structures that we'd need for handling a basic example of successive-cyclic *wh*-movement.[1]

(7) What$_i$ did John t$_i$ say t$_i$?

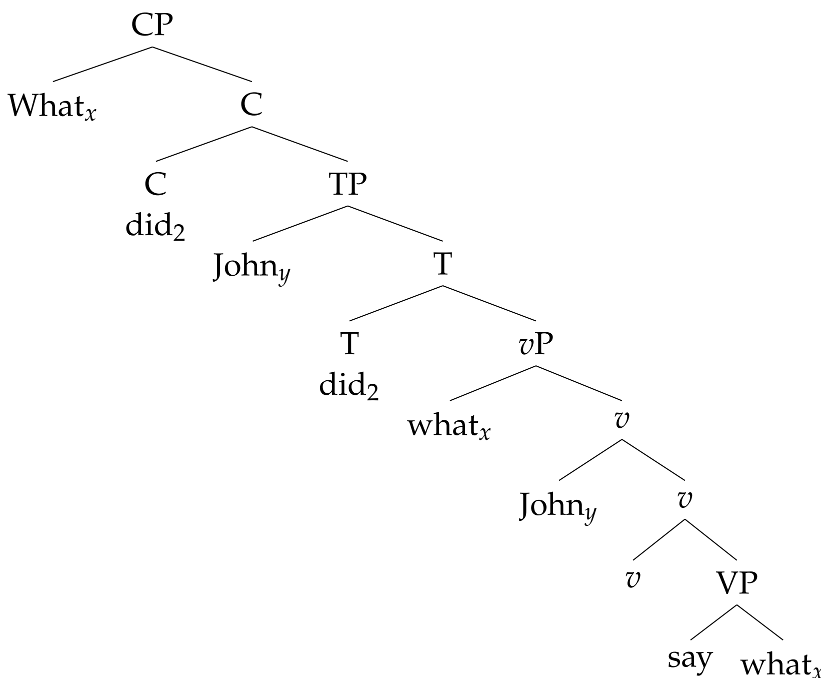
a.



b.



c.



[1]Note that the indices aren't really there, so they don't get in the way of Match. They're just there to show how the instances of *what* and *John* in the two phases come to be unified upon Stitch.

5 Some consequences of Phase Stitching

5.1 Left-to-right derivations

An interesting and potentially useful property of Stitching follows from the full modularity of the derivation of individual phases:

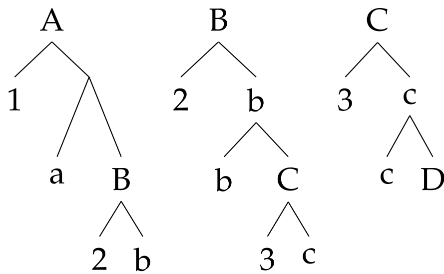
☞ Under Transfer, phases are built up sequentially in a cyclic fashion. But with Stitching, phases are built up completely independent of each other in their own workspaces.

☞ So, while in Transfer higher phases depend on the output of lower ones, and hence lower ones must be constructed first, in Stitching the order in which phases are constructed is irrelevant.

☞ Furthermore, the order in which phases are Stitched together is irrelevant, because the way their edges interact essentially depends on static matching.

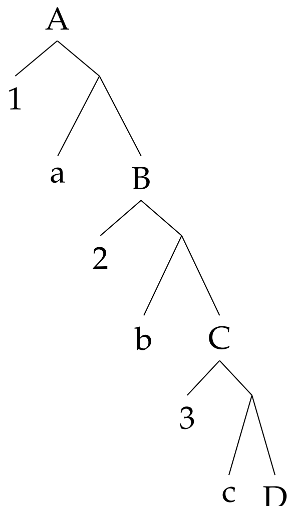
I should clarify that I do not mean **linear** or **hierarchical** order here, but **derivational** order. Take e.g. 3 phases with the structures in 8:

(8)



- Their shapes along with 4 will ensure that B is stitched into A, and C is stitched into B yielding 9:

(9)



- No other combination — e.g. B stitching into C or C stitching into A — will be possible, because the relevant edges don't Match. So the resulting hierarchical order, and the linear order derived from it, are relevant and pre-determined.

- However, it is irrelevant whether we first stitch C into B, and then B into A, or the other way around. Both derivational orderings will yield the same structure in 9.

This is important because it allows us an interesting approach to a well-known tension about directionality of derivations:

☞ On the one hand, we have evidence from things like basic compositionality and cyclicity effects that derivations proceed in a bottom up fashion.

☞ But we also have evidence from production and parsing that performance must operate to a large extent in a left-to-right fashion (see Phillips 2003, Phillips and Lewis 2013, for some arguments).

☞ Bottom up and left-to-right aren't exactly opposites, but they are incompatible with each other, so we have an apparent inconsistency.

Given what we've just said, however, Stitching has the potential to resolve this tension:

- Derivational steps within a single workspace, i.e. inside a phase, are bottom up.
- Stitching phases together, however, can proceed left-to-right.
- We can then construct our theories of competence and performance to take advantage of this distinction, modeling particular phenomena according to which kind of directionality effects they show.

Note that one potential interpretation of all of this is that Stitch isn't strictly speaking a derivational operation, but rather a constraint on well-formed representations.

☞ It is not accidental that we see an overlap here with arguments for "constraint-based" approaches (Jackendoff 2011, Müller 2013) on the basis of their ability to model performance.

5.2 Triggering intermediate movement

Stitching also allows a better story for a common concern of phase-based work: the treatment of intermediate steps of successive-cyclic movement. Here's a quick description of the issue, based on 10, simplified to ignore *v*P as a phase:

(10) *Who* did Stringer think [$_{CP}$ <*who*> that Omar shot <*who*>]?

- We assume that the matrix C has a feature on it that drives a *wh*-element to its specifier, because this is clearly a syntactic requirement of English *wh*-interrogative clauses.
- It can't get there directly from its base position, which will be shipped off to Spell-out before matrix C enters the derivation.
- This problem can be overcome if it moves successive-cyclically via the lower Spec-CP which, being in the edge, will be visible to the next phase up.

But what actually makes *who* move to the Spec of the lower CP on its way up?

☞ The embedded CP does not have interrogative force, so in and of itself it has no requirement to have a *wh*-element in its specifier. In fact, it can't really have one end up there:

(11) * Stringer thinks [$_{CP}$ *who* (that) Omar shot <*who*>]

☞ It's clear that the higher Spec-CP wants to force movement, and that that movement will be blocked if *who* doesn't move to the intermediate Spec-CP on the way.

☞ But given the logic of phase theory, the higher C won't yet have entered the structure at the point when this movement would have to be triggered!

➪ So we have to do something clever to trigger intermediate movement in case it is needed, but make sure that we don't get it in cases where it is not needed.

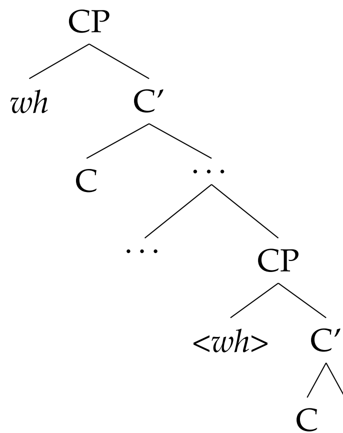
There are various ideas out there about how to do it, but most boil down to some version of 'overgenerate and filter':[2]

- We assume some mechanism that can trigger movement in contexts where it is not locally required, and make this somehow be optional.
- We make sure that there are constraints to rule out structures with the wrong combination of upper and lower portions — e.g. an interrogative main clause and a lower clause without intermediate movement, or a declarative main clause and a lower clause with intermediate movement.
- We then generate all conceivable combinations, and the ill-formed ones get filtered out at the interfaces for having unchecked uninterpretable features or the like.
- This does the basic job of course, but seems to suggest a waste of computational resources, as a large subset of derivations (the (vast) majority?) will lead to a crash.

I think Stitching gives us a fairly satisfactory way to implement the idea behind overgenerate-and-filter without actually overgenerating:

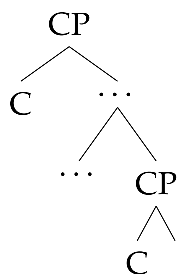
☞ Higher phases with *wh*-movement landing sites (where the base position of the *wh*-element isn't local) will always have a copy of the *wh*-element in their inner edge, i.e. they'll look like this:

[2] Heck and Müller (2000), Müller (2011) have an approach that doesn't really fit into the overgenerate and filter category, based on the idea of phase balance. If I understand correctly, though, phase balance requires access to a numeration for the entire sentence, not just information about the current phase, so that, in a sense, while you can't look ahead to the structure that's going to be built, you can look ahead at what elements are going to be involved.

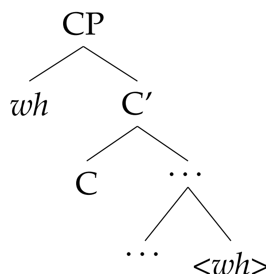


☞ So they will only be able to Stitch with lower phases that also have such an element in their outer edge. I.e. Stitch will simply not happen with a structure without intermediate movement.

☞ Higher phases without a *wh*-movement landing site will **not** have a copy of a *wh*-element in their inner edge, i.e. they'll look like this:



☞ So they will only be able to Stitch with lower phases that also have an outer edge without a *wh*-element in the specifier. I.e. Stitch will simply not happen with a structure like the one below with intermediate movement:



Here's how it has to work:

- We set things up so that Merge and Agree can create all of the phase types that will actually be needed in convergent sentences of the language.
- This (in part) amounts to saying that, for every phase generated with an internal edge, at least one phase will also be generated with a matching external edge.

- Then we don't have to actually overgenerate and filter. We generate a number of individually convergent phase pieces, and then choose the appropriate ones to Stitch together.
- Non-convergent structures (at least of the kind relevant here) can never be generated, so there's no need to filter them out.

5.3 CED effects

One last reason why Stitching is worth pursuing is that it straightforwardly derives traditional CED effects (see Huang 1982, Müller 2011, and lots of other stuff in between), in a way that's parallel to Uriagereka (1999).

(12) **Condition on Extraction Domains**
You can only move things out of complements, i.e. not out of specifiers or adjuncts.

Here are some quick examples demonstrating the basics:

(13) a. Rhonda likes [drinking water].
b. What does Rhonda like [drinking <*what*>]? **(✓From complement)**

(14) a. [Drinking water] makes Rhonda thirsty.
b. * What does [drinking <*what*>] make Rhonda thirsty. **(*From specifier)**

(15) a. Gary shouted [that Mary brought beer].
b. What did Gary shout [that Mary brought <*what*>]? **(✓From complement)**

(16) a. Gary shouted [because Mary brought beer].
b. * What did Gary shout [because Mary brought <*what*>]? **(*From adjunct)**

- Now, once you start looking at different kinds of subjects, complements and adjuncts, and different languages, things get way more complicated than 12 would suggest.
- For this talk I'm just going to ignore all of that and assume that it's roughly correct as a descriptive generalization.

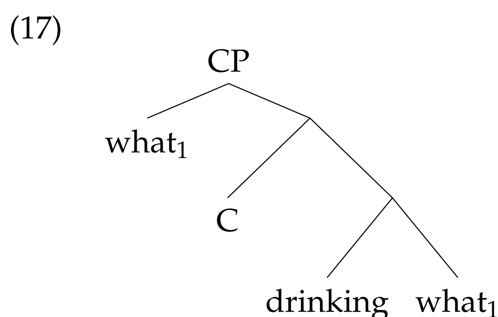
First of all, Stitching implies that all specifiers and adjuncts must be phases:

☞ They are complex left branches, thus must be built in their own Workspace, thus must be phases.

Now here's what happens if we try to extract out of a specifier:

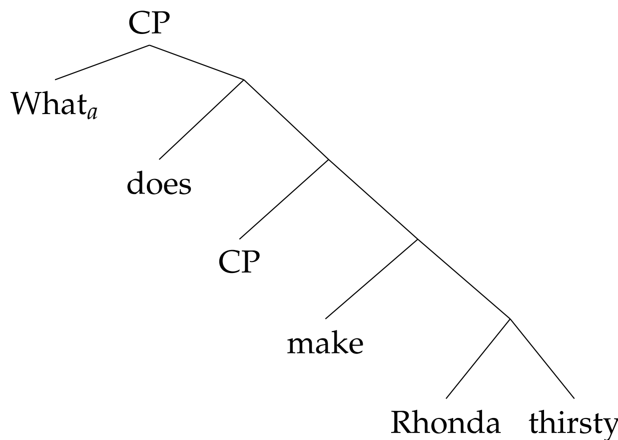
- We can build up the subject phase and move the *wh*-element to its edge without any trouble (I'm assuming for concreteness that this clause is a CP and ignoring the interior details beyond that, but its category actually makes no difference here):

(17)



- Then we build the matrix phase as follows:

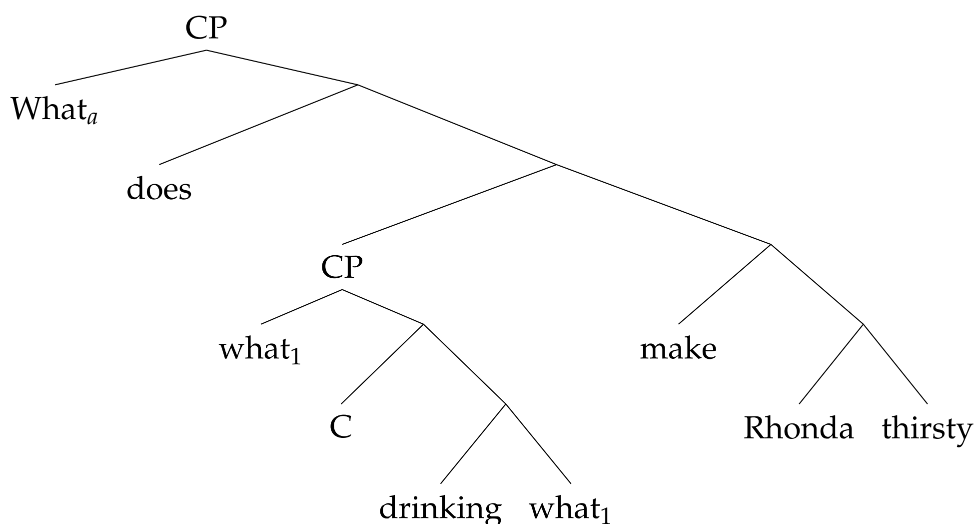
(18)



Note that the representation of the clausal subject here has to be a single non-branching node because of the way our Workspaces ban complex left branches. I.e. we get simultaneously maximal and minimal CP projection which can include only featural information – its category, maybe *wh*-features and the like, but crucially no actual internal structure.

- Now we can perhaps Stitch 17 into that subject CP position, since their visible edges Match — i.e. they're both C projections. This would give us the following:

(19)



- But notice that the *what* in the matrix Spec-CP is **not** unified with the two *what* copies in the subject clause. This is because they were not in Matching Edges that Stitch has applied to, precisely because the representation of the matrix clause had no representation of the internal structure of the embedded clause.

- Since such unification through Stitch on Matching Edges is the only way to derive dependencies between elements in two different phases, what we have here is not actually extraction. And indeed, the result will be ruled out, e.g. because the *what* in the matrix clause can't get a θrole.

Adjuncts will have the same problem, although with them it's perhaps even worse:

☞ Being unselected, it's plausible to think that they won't even be represented as minimal non-branching edges in their matrix clauses before Stitching takes place.

Again, there's far more to be said about CED effects, and about the generally worrisome status of adjuncts, but I'll leave it at that for now.

References

Chomsky, Noam. 2000. Minimalist inquiries: the framework. In *Step by step: Essays on minimalism in honor of Howard Lasnik*, ed. Roger Martin, David Michaels, and Juan Uriagereka. Cambridge, Mass.: MIT Press.

Chomsky, Noam. 2001. Derivation by phase. In *Ken Hale: A life in language*, ed. Michael Kenstowicz. Cambridge, Mass.: MIT Press.

Chomsky, Noam. 2008. On phases. In *Foundational issues in linguistic theory*, ed. Robert Freidin, Carlos Otero, and Maria Luisa Zubizarreta, 133–166. Cambridge, Mass.: MIT Press.

Hale, Mark. 2007. *Historical linguistics: Theory and method*. Oxford: Blackwell.

Heck, Fabian, and Gereon Müller. 2000. Successive cyclicity, long-distance superiority, and local optimization. In *Proceedings of WCCFL 19*, 218–231.

Huang, C.T. James. 1982. Logical relations in Chinese and the theory of grammar. Doctoral Dissertation, MIT.

Jackendoff, Ray. 2011. What is the human language faculty? two views. *Language* 87:586–624.

Müller, Gereon. 2011. *Constraints on displacement. a phase-based approach*, volume 7 of *Language Faculty and Beyond*. Amsterdam: Benjamins.

Müller, Stefan. 2013. Unifying everything. *Language* 89:920–950.

Phillips, Colin. 2003. Linear order and constituency. *Linguistic Inquiry* 34:37–90.

Phillips, Colin, and Shevaun Lewis. 2013. Derivational order in syntax: Evidence and architectural consequences. *Studies in Linguistics* 6:11–47.

Uriagereka, Juan. 1999. Multiple spell-out. In *Working minimalism*, ed. Samuel Epstein and Norbert Hornstein. Cambridge, Mass.: MIT Press.