Combinatory Categorial Grammar and Link Grammar are Equivalent

Linas Vepstas

Updated 14 July 2022

Abstract

This is a short, semi-formal note explaining how Combinatory Categorial Grammar (CCG) and Link Grammar (LG) are equivalent. It covers some basic ideas from proof theory, type theory, and the "sexuality" in type combinators. The key idea exposed is that type theory must be combined with connector sexuality in order to get a fully general framework encompassing proof theory, logical inference, and linguistics that is also free of ambiguities and implicit assumptions.

A Question posed on a Discord chat channel

@Adam Vandervorst asks: *Does anyone here know about https://en.wikipedia.org/wiki/Combinatory_categorial_grammar?*

> (*from Wikipedia*) Combinatory categorial grammar (CCG) is an efficiently parsable, yet linguistically expressive grammar formalism. It has a transparent interface between surface syntax and underlying semantic representation, including predicate–argument structure, quantification and information structure. The formalism generates constituency-based structures (as...

I talked to its inventor last week (who has since moved on to do learning in linguistics) and it was really interesting — Today at 9:31 AM 13 July 2022

Prelude

The difference between physicists and mathematicians is that the physicists don't sweat the details. They get to say things like "ah, whatever, I get the general idea, let's just assume this is right, and proceed, and see where we get." Losing months to prove a minor detail that seemed like it was right all along is just ... a loss of time. Physicists focus on those details that seem to be wrong or inconclusive. That's where you dig for gold.

This paper is a physics paper, not a math paper. I beleive everything in it is absolutely correct; however, there are no formal proofs here. There are proofs in here, they're just not formal.

The Nature of Grammar

To open, there's this thing about grammars that you should know.

As far as I can tell, all of the different (formal) grammar formalisms are interconvertible into one-another, by purely algorithmic means. That is, given the collection of symbols and rules that are used to define one formalism (*e.g.* constituency grammars, CG) one can convert that into a different formalism (*e.g.* dependency grammars, DG) by applying a purely automatic transformation on the grammar specifications. No hand-waving is required, nor any metaphysics: a machine can convert DG into CG and *vice versa*, and that machine is rather simple.

Somewhere out there is a nice paper that explains how to convert between DG and CG and back. It provides a simple algo to do this. Sadly, I have misplaced the reference. A paper by Xia and Palmer[1] is not quite as extensive, but gives a flavor of the idea.

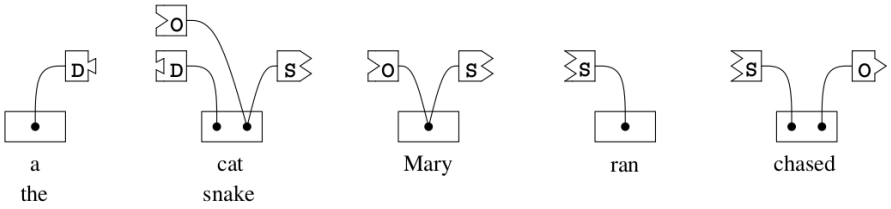
CCG is Equivalent to LG

I looked at CCG many years ago, and from what I could tell, for each and every CCG compound type, one has an equivalent LG link type, and *vice versa*. For example, the compound type *NP/N* is the same thing as the LG `D+` link (determiner) type and *(S\NP)/NP* is the just the LG `S- & O+` (verb taking subject and object) and one can march down the list this way. The goal of this PDF is to make the above statement precise.

At first, it's mildly confusing, because it seems like the compound type *NP/N* might be encoding some kind of structure that the single-letter, single-type `D+` is not ... but, actually, no, that is incorrect. The CCG notation is not actually "more atomic" or "more compositional" than the LG notation. To understand this, one must slightly shift one's point-of-view.

Jigsaw Pieces

Recall how I talk about "jigsaw pieces" all the time? Some example LG jigsaw pieces:



The above diagram is taken from the original 1991 paper presenting Link Grammar.[2] Now, lets look at that CCG Wikipedia article. You can find this inference rule:

$$\frac{\alpha : X/Y \quad \beta : Y}{\alpha\beta : X} >$$

This says that (roughly speaking) "if you have a jigsaw called alpha and it has connector of type X on left and type Y on right, and if you have jigsaw beta with a connector Y, you can connect the two Y's together, to yield a combined jigsaw alphabeta having only one unconnected connector X."

Lets now try to be more precise. This inference rule uses conventional proof-theory style notation. The horizontal line is the defines the inference to be done. Above the line are the inputs, below the line are the outputs. The Greek letters α, β are terms, and the Latin letters X, Y are types. The colon indicates a term-type pairing, so that $\beta : Y$ is a term of type Y. The slash $/$ and the backslash $\backslash$ are type constructors, so that X, Y are simple types, and X/Y is a compound type, constructed from the two simpler types. The $>$ is just a label for the rule; it has no syntactic role.

The CCG Wikipedia article calls these inference rules "combinators". Above is one "application combinator"; there is also a second rule:

$$\frac{\beta : Y \quad \alpha : X\backslash Y}{\beta\alpha : X} <$$

Lets rewrite these two rewrite rules in LG notation. They would be

$$\frac{\alpha : (X- \ \& \ Y+) \quad \beta : Y-}{\alpha\beta : X-} >$$

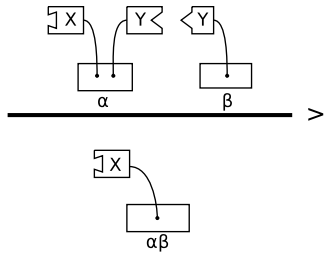
and

$$\frac{\beta : Y+ \quad \alpha : (X+ \ \& \ Y-)}{\beta\alpha : X+} <$$

Here, the X, Y are the LG types, called "link types" in the literature. The $Y+$ and $Y-$ are called "connectors": they are jigsaw-puzzle-piece tabs, as-yet unconnected. When they do connect, they are called a "link", and thus the name "Link Grammar". The $+$ and $-$ are the "connector directions": they specify in which direction a connector can connect: to the right or to the left.

The ampersand & is a "kind of" type constructor. Given two connectors, say, $X-$ and $Y+$ it creates a new type (more precisely, a "jigsaw") $X- \ \& \ Y+$. This can be made more precise, in an upcoming section.

So what are these two inference rules really saying? Well, its almost trivial: they're just saying "connectable connectors can connect, if the connector types are identical, and the sexuality of the connectors is opposite." Let's cement the obvious. Here's the first combinator, using the same diagrammatic representation as in the original 1991 Link Grammar paper:



What is this picture saying? The obvious: when you combine terms α and β the result is a single term $\alpha\beta$ and it is convenient to not draw, to ignore, to pretend that the link Y joining these two pieces as disappeared. In other words, a partially-assembled jigsaw puzzle behaves exactly like a single jigsaw piece.

In the above, the type Y is meant to be understood as a primitive type, an ur-type belonging to the theory. The situation for types Y that are compond types, built with type constructors, requires a bit of finesse. This is discussed further below, after the examples.

The CCG Composition Combinators

For completeness, the remaining CCG combinators should be treated as well. Here's a side-by-side Rosetta Stone of the two composition combinators.

CCG	LG
$\frac{\alpha{:}X/Y \quad \beta{:}Y/Z}{\alpha\beta{:}X/Z} B_>$	$\frac{\alpha{:}X- \,\&\, Y+ \quad \beta{:}Y- \,\&\, Z+}{\alpha\beta{:}X- \,\&\, Z+} B_>$
$\frac{\beta{:}Y\backslash Z \quad \alpha{:}X\backslash Y}{\beta\alpha{:}X\backslash Z} B_<$	$\frac{\beta{:}Y+ \,\&\, Z- \quad \alpha{:}X+ \,\&\, Y-}{\beta\alpha{:}X+ \,\&\, Z-} B_<$

Clearly, they just specify how to connect single connectors on compound jigsaws.

The CCG Type-raising Combinators

The last pair of combinators are the type-raising combinators. These are

CCG	LG
$\frac{\alpha{:}X}{\alpha{:}T/(T\backslash X)}T_>$	$\frac{\alpha{:}X-}{\alpha{:}T- \,\&\, T+ \,\&\, X-}T_>$
$\frac{\alpha{:}X}{\alpha{:}T\backslash(T/X)}T_<$	$\frac{\alpha{:}X+}{\alpha{:}T+ \,\&\, T- \,\&\, X+}T_<$

The interpretation of these two rules is that, given a single (assembled) jigsaw X, cut it into two (disassemble it) such that the new connectors are of type T (and they are no longer connected).

There seems to be a slight awkwardness, as the earlier combinators could be easily understood by thinking only about simple types. By contrast, the type-raising combinator requires a more complex explanation:

> *"The type-raising combinators, often denoted as $T_>$ for forward type-raising and $T_<$ for backward type-raising, take argument types (usually primitive types) to functor types, which take as their argument the functors that, before type-raising, would have taken them as arguments."*

Phew. That's a mouthful, when all that is really being said is "disconnect" or "cut into pieces". An extensive discussion on the linguistic interpretation and linguistic utility of type-raising is given in a longer section, below.

An Sloppy Example

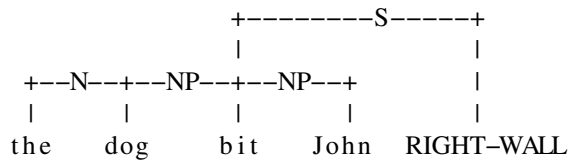
The Wikipedia article includes an example of two different proofs (two different derivation trees) of the same sentence. The sentence is "*the dog bit John*". Here's one derivation tree:

$$\dfrac{\dfrac{\dfrac{\text{the}}{NP/N}\quad\dfrac{\text{dog}}{N}}{NP}> \quad \dfrac{\dfrac{\dfrac{\text{bit}}{(S\backslash NP)/NP}\quad\dfrac{\text{John}}{NP}}{S\backslash NP}>}{S}<$$

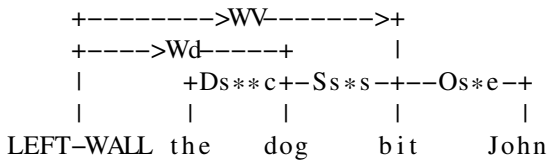
If we are sloppy and uncareful ***, we find the translated LG derivation rules:

$$\dfrac{\dfrac{\dfrac{\text{the}}{NP+\ \&\ N+}\quad\dfrac{\text{dog}}{N-}}{NP+}> \quad \dfrac{\dfrac{\dfrac{\text{bit}}{S+\ \&\ NP-\ \&\ NP+}\quad\dfrac{\text{John}}{NP-}}{S+\ \&\ NP-}>}{S+}<$$

This is perhaps hard to read? The conventional LG notation for this derivation would be:



where an extra jigsaw piece `RIGHT-WALL: S-` was introduced, so as to keep all connectors fully connected. The above works. It is not the preferred LG parse for the current English language dictionary. That would be:



The link types are obviously more complex. Note also the present of a cycle (the triangle, whose edges are `WV`, `Wd` and `Ss*s`.) Note the presence of several directed connectors. The complex upper-case/lower-case link types are an example of "sexuality"; see next section.

A Less Sloppy Example

*** Wait, what? Sloppy and uncareful? If we are careful, and don't gloss any plus and minus signs, then the following derivation results:

$$\dfrac{\dfrac{\dfrac{\text{the}}{NP- \;\&\; N+}\quad\dfrac{\text{dog}}{N-}}{NP-}> \quad\dfrac{\dfrac{\text{bit}}{S+ \;\&\; NP- \;\&\; NP+}\quad\dfrac{\text{John}}{NP-}}{S+ \;\&\; NP-}>}{NP- \;\&\; S+ \;\&\; NP-}\ \text{fail !!}$$

This reveals a bug in the Wikipedia article derivation. It should have been:

$$\dfrac{\dfrac{\dfrac{\text{the}}{N}\quad\dfrac{\text{dog}}{NP\backslash N}}{NP}< \quad\dfrac{\dfrac{\text{bit}}{(S\backslash NP)/NP}\quad\dfrac{\text{John}}{NP}}{S\backslash NP}>}{S}<$$

This is provides a hint as to why LG might actually be better than CCG: it's easier to spot bugs. We live in an era of compilers and debuggers; yet hand-writing expressions is error prone.

Constituency and Dependency

The reason for this bug appears to be a slavish adherence to the conventions of olde-fashioned constituency grammar. The inherited tradition is that N denotes a noun, and

NP is a noun phrase. When one writes *"The dog bit John"*, it is clear that "*the dog*" is an NP, and its also clear that "*the*" is not N, and that "*dog*" is N. Thus, one is forced into assigning NP/N to the determiner. But this is an error!

The markup NP/N is saying that the word "*the*" is a noun-phrase, and it's just missing a noun before it becomes a complete NP. Do you really want to give such a primal ascendancy to the word "*the*"? It makes it the head of a head phrase. Hard to imagine that determiners are head words.

Knowing even a little of dependency grammar would have exposed the error: "*the*" should have been D and "*dog*" should have been N (if it stands alone) or NP\D (if its a noun taking a determiner). But conventional constituency grammars rarely if ever bother with issuing a distinct type for determiners, and thus we arrive at a basic markup error. The road to hell is indeed paved with Chomskian gold.

A Second Example

The CCG article also gives an alternative derivation for the sentence. It is also problematic. The article currently states:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{the}}{NP/N}\quad\dfrac{\text{dog}}{N}}{NP}>}{S/(S\backslash NP)}T_{>}\quad\dfrac{\text{bit}}{(S\backslash NP)/NP}}{S/NP}B_{>}\quad\dfrac{\text{John}}{NP}}{S}>$$

Translating this to into the LG combinator form reveals an issue, apparently with the $B_>$ rule:

$$\dfrac{\dfrac{\dfrac{\dfrac{\text{the}}{NP- \;\&\; N+}\quad\dfrac{\text{dog}}{N-}}{NP-}>}{S- \;\&\; S+ \;\&\; NP-}T_{>}\quad\dfrac{\text{bit}}{S+ \;\&\; NP- \;\&\; NP+}}{S- \;\&\; S- \;\&\; NP- \;\&\; S+ \;\&\; NP-}B_{>}\quad\dfrac{\text{John}}{NP-}}{XXX}\ \text{fail !}$$

There does not appear to be any fix for this, while continuing to employ the $B_>$ rule. We can get rid of the excess of S's by getting rid of the $T_<$ inference, and replacing the $B_>$ inference by $<$ and so writing

$$\dfrac{\dfrac{\text{the dog}}{NP+}\quad\dfrac{\text{bit}}{S- \;\&\; NP+ \;\&\; NP-}}{S- \;\&\; NP+}<$$

Working backwards to get the CCG form, this becomes

$$\dfrac{\dfrac{\text{the dog}}{NP}\quad\dfrac{\text{bit}}{(S/NP)\backslash NP}}{S/NP}<$$

From this point, the rest of the derivation can go through, as before.

Primitive Types vs. Compound Types

A closer look at the $B_>$ rule reveals an issue with the LG mapping. The rule is this:

$$\frac{\alpha : X/Y \quad \beta : Y/Z}{\alpha\beta : X/Z} B_>$$

In the original LG mapping, the type Y was taken to be a primitive type. It would be one of the inbuilt types of the system, and not one that was constructed by means of type constructors. Yet, in the second example, $B_>$ is being applied with $Y = (S\backslash NP)$ which is a compound type, not a primitive type. Perhaps the translation to LG was flawed? Let's look at it again. The relevant part is

$$\frac{\dfrac{\text{the dog}}{S/(S\backslash NP)} \qquad \dfrac{\text{bit}}{(S\backslash NP)/NP}}{S/NP} B_>$$

and so perhaps the translation should have been

$$\frac{\dfrac{\text{the dog}}{S- \ \& \ (S\backslash NP)+} \qquad \dfrac{\text{bit}}{(S\backslash NP)- \ \& \ NP+}}{S- \ \& \ NP+} B_>$$

and *now* the two compound types can connect to one another, correctly. Does this insight provide the correct LG mapping, finally? Well it depends...

One possibility is to create a brand-new primitive LG type, call it *SU* for "subject", and employ the mapping

$$(S\backslash NP) \mapsto SU$$

Then the above translation goes through perfectly well, and the $B_>$ rule can be kept as-is.

Another possibility is to attempt to work with the compound type. This will not work, and here's why. The LG mapping is

$$(S\backslash NP) \mapsto S+ \ \& \ NP-$$

and so perhaps one could in infer that

$$(S\backslash NP)- \mapsto S- \ \& \ NP+$$

But this will not work, because writing $S+ \ \& \ NP-$ to the left of $S- \ \& \ NP+$ does not allow it to be contracted. The S parts can be contracted, because $S+$ is to the left of $S-$ but the NPparts cannot be contracted – they are facing away from each other.

To conclude: the mappings from the six CCG combinators to the equivalent LG combinators work if they are also supplemented with additional mappings from compound CCG types to primitive LG types. If they are not supplemented, then inference paths that require compound types to appear in the combinators must be avoided.

Equivalence, or Not?

In order to preserve the equivalence of CCG and LG by means of the straight-forward translation of the inference rules, (*i.e.* avoiding the compound types) two changes had to be made to the second example: The use of the type-raising rule $T_<$ had to be abandoned, and the form for the verb had to be changed into

$$\frac{\text{bit}}{(S/NP)\backslash NP}$$

which is *not* the same form as that in first example.

Is this too much to ask for? Is it OK to say that, sometimes transitive verbs have the form $(S\backslash NP)/NP$ and sometimes they have the form $(S/NP)\backslash NP$? Certainly LG doesn't care: LG just provides one connector going left, to the subject of the verb, and another going right, to the object. To support two homotopic parse trees in CCG, should there be one more inference rule, say, for example, a limited associativity rule:

$$\frac{\alpha : (X\backslash T)/T}{\alpha : (X/T)\backslash T}\ \text{WAssoc.}$$

or even a broad one:

$$\frac{\alpha : (X\backslash Y)/Z}{\alpha : (X/Z)\backslash Y}\ \text{SAssoc.}$$

One has four possibilities, then:

1. Transitive verbs can be written in either of two forms: $(S\backslash NP)/NP$ or $(S/NP)\backslash NP$.
2. Transitive verbs only have one form, but an associativity rule provides homotopic equivalence.
3. The equivalence between LG and CCG is a false mirage; the proposed fixes must be rejected, and the use of the $T_<$ rule is just fine, as it stands.
4. The equivalence between LG and CCG is true, but an LG primitive type has to be introduced for any CCG compound type appearing in a reduction.

Hard-core adherents to CCG may opt for the third case, and live in denial. But careful if you chose this option: it is glossing over a deeper interpretational issue concerning types, connectors and connector sexuality. This is delved into much greater detail in an upcoming section. The crux is that in many situations, when people say "type", they really mean "connector", but leave the connector direction (polarity, sexuality) $+/-$ implicit, deducible from context. The default presentation of CCG assumes mono-sexual types (types without the $+/-$ directional markup), and makes implicit assumptions about polarity, left to the reader to infer from context. This is dangerous: leaving implicit, unstated conventions to the reader to blithely assume is just asking for trouble. I think we've found trouble, here.

The root cause of both of these bugs was a failure to attend the polarity that is implied by the type constructors $/$ and $\backslash$. These type constructors build compound types with an implicit polarity; the failure to write it down leads to interpretational

issues. These bugs can only be resolved by taking care to distinguish between types and sexualities (polarities, here, since the sexualities here are heterosexual.) More on sexuality, shortly.

Homotopic Equivalence

There is yet another infelicity in the Wikipedia article. It currently states:

> *The sentence "the dog bit John" has a number of different possible proofs. Below are a few of them. The variety of proofs demonstrates the fact that in CCG, sentences don't have a single structure, as in other models of grammar.*

This is misleading. Two different derivation trees are presented. The ultimate parse is identical. This phenomenon is commonly treated in textbooks on proof theory: two different proofs have proof trees that appear to be different, but can be rearranged by homotopic deformations into one-another.[3] That is, there is a Scott-continuous deformation, referring to the Scott topology that conventionally applied to proofs/programs.

How can continuous transformations be spotted? This is out of bounds for the current text; however, a taste of that flavor can be gotten from the associativity inference rule, above. Roughly speaking, one proceeds at a meta level, by indicating when two inferences are equivalent, effectively by an associativity (meta-)rule.

Free Object Lemma

Are the mappings of the six combinators sufficient to prove equivalence? In category theory, there is a lemma, we'll call it the "free object lemma", that says homomorphisms extend "trivially" to free objects. That is, if one has a homomorphism $A \to B$ and $F(A)$ is the free object on A, then the homomorphism extends to $F(A) \to F(B)$.

In the present case, the mapping $A \to B$ is the mapping of the six combinators. We haven't really "proven" that the mapping is truly a homomorphism, preserving all algebraic properties of the combinators. Instead, we did a hand-wavey "see this makes perfect sense" kind of argument. Is this enough? Well, for the CCG primitive types, there does not seem to be any further structure or algebraic properties to consider, and so the presented mapping is trivially a homomorphism. For the case where compound types appear in the combinators, we've discovered, by way of example, that each CCG compound type must be mapped to a new LG primitive type, before it can be properly reduced (connected). In other words, compound types in CCG do have non-trivial algebraic properties, and these must be "forgotten" with the mapping. Exercise left to the reader to restate the same thing, using the forgetful functor.

The second part of applying this lemma is to confirm that CCG really is the free object of the six combinators. I beleive the answer is yes. There are not any further constraints on combining the combinators: all possible syntactically-valid combinations are valid.

Moving in the opposite direction, from LG to CCG, is less obvious. The LG disjuncts do have an algebraic property that must be respected: LG connectors are commutative, when the polarities differ. For example, $(X+ \ \& \ Y-) = (Y- \ \& \ X+)$. Yet the mapping from LG to CCG is that $(X+ \ \& \ Y-) \mapsto X \backslash Y$ whereas $(Y- \ \& \ X+) \mapsto Y/X$, and the two are inequivalent in CCG. In either case, in CCG, Y connects to the left, and X connects to the right. Perhaps the resulting language (the free object) is exactly the same, but this is not immediately obvious.

In essence, LG doesn't care about the order in which connectors are connected during parsing, while CCG does. In CCG, when connectors are joined in a different order, a different derivation tree results. It is reasonable to argue that these two should be homotopic, and that there should not exist an obstruction preventing connection sequence reordering. To repeat: $(X+ \ \& \ Y-) = (Y- \ \& \ X+)$ is a statement about Scott-continuity. Can we treat equality as equivalence? That is, is the free theory of CCG modulo commutativity equivalent to the free theory of LG? I think so, because that is how the free object works, in general, when one has modulo constraints that commute with the homomorphism.

Type-Raising and Link Crossing

The fix proposed to the example above eliminates the use of the type-raising rule. It's an interesting rule, but when is it actually needed, linguistically speaking? LG does not explicitly have such a rule; if more connectors are needed on some particular grammatical class, one can simply put them there. What is the linguistic significance of the type-raising combinator? How does it impact the equivalence of CCG to LG? The answer to these questions is that it enables link crossing, and equivalence is preserved.

LG has a global rule, and that is that all parse graphs must be planar. In other words, two links may not cross. There are a number of reasons for this global rule:

1. Linguists have determined that most languages do not need parses that involve link-crossing. There are exceptions (Finnish, Turkish?, ...) but these seem rare.

2. Studies in psycholinguistics shows that sentences with crossing links are more difficult to comprehend. Humans take longer to understand such sentences.

3. The planarity constraint enables certain kinds of parsing algorithms that are not possible for the general non-planar case. Those algorithms run several orders of magnitude faster, on ordinary-sized sentences. This has a practical impact on real-world software.

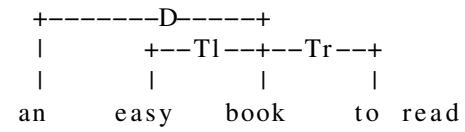
4. Despite a global planarity constraint, there is a work-around: with appropriate connectors, one link can be passed through another. The trick is similar to that of drawing non-planar electric circuits on a flat sheet of paper: one simply draws "hops" where they are needed.

5. This is a good thing, because for English, there are examples where crossing links are appropriate and needed.

Some examples:

- "*An easy book to read*" requires the links *"an—book*" and "*easy—to*"; these cross. (It is the reading that is easy, not the book.)
- "*It was announced that remains have been found of the ark of the covenant.*" requires links "*that—have*" which crosses "*remains—of*". (The head verb of the subordinate verb phrase is "*have*", and therefore "*that*" must link to the head.)

In the face of the planarity constraint, link crossings can be engineered in LG by designing jumping connectors. Suppose a link of type `T` connecting "*easy—to*" has to cross a link of type `D` connecting *"an—book*". Then one invents new link-types `Tl` and `Tr` (left and right) and a dict entry "`book: Tl- & D- & Tr+;`". Diagrammatically, this works out as:



LG could have a rule (but it doesn't) that could globally enable link-crossing, in all cases. It would take any dictionary entry "`word: X+;`" and convert it to the entry "`word: X+ or (Yl- & X+ & Yr+);`" for any link types `X` and `Y`. It does not have such a global rule, because of the reasons spelled out above: in practice, link-crossings are rare, and, as a general rule, one wants to tightly control them. Some link types can be allowed to cross, others can be prohibited. One might even want to control which type is the crosser: in the above example, it could have been arranged so that the `D` link splits in two, instead of the `T` link.

Hopefully, the above discussion has made clear what the type-raising combinator really is: it is a link-crossing enabler. That is, the rule

$$\frac{\alpha : X}{\alpha : T/(T\backslash X)} T_{>}$$

says, in plain terms: "given any connector to type X, keep the X, but surround it by two new connectors, one linking to the left, and one linking to the right. The two new connectors must have exactly the same type T." Just to hammer this home, here it is again, as an LG rule:

$$\frac{\alpha : X-}{\alpha : T- \ \& \ T+ \ \& \ X-} T_{>}$$

and, since connectors of opposite polarity commute in LG, this could be written as

$$\frac{\alpha : X-}{\alpha : T- \ \& \ X- \ \& \ T+} T_{>}$$

Again, LG eschews such a rule, except for a certain limited set of types T. How should the CCG type-raising combinators be understood? Are they combinator classes, with one such combinator for any and every type T, or is the intent to limit it to only certain types T? If the former, then LG and CCG are not equivalent, since LG does not include such a global rule. If the later, then yes, the two grammars remain equivalent.

Conclusion

In conclusion: CCG is equivalent to LG. The inference rules of CCG are merely rules for how to join together connectors. Two rules connect simple types to compound types; two more rules connect compound types, and the final two rules show how to disassemble connections (equivalently, to create unconnected pairs).

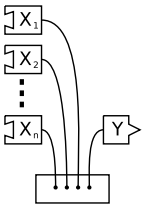
It should be clear that CCG uses a far more awkward notation (the proof-theoretical inference-bar notation). Awkwardness matters, because concepts like link-crossing and Dick Hudson's "landmark transitivity" become hard to talk about in CCG.

Proof Theory

Although the presentation above focused on CCG, and LG, the concept of inference rules as being certain peculiar kinds of rewrite rules is not new. Lets take a look at the "standard form" of an inference rule, taken from Wikipedia:

$$\frac{\begin{array}{c}\text{Premise \#1}\\ \text{Premise \#2}\\ \cdots\\ \text{Premise \#n}\end{array}}{\text{Conclusion}}$$

This is, oddly enough, just another jigsaw. Let's be painfully clear, by actually drawing it:



The X_k are the premises, the Y is the conclusion. These are drawn as if they're typed. The jigsaw connector shapes are just illustrative; what matters in this picture are the connector directionalities: there are n inputs and one output. Structurally, this has the form of a lambda combinator, having n inputs ... in practical applications, inference rules behave as if they were lambdas. The central point being made here is that the input-to-output connections are heterosexual. Premises cannot be "plugged into" premises; conclusions cannot be "plugged into" conclusions. There is only one possible direction: conclusions can be plugged into premises, and nothing more.

All proof-theoretical inference rules are always jigsaw pieces. All of them, without any exceptions. This holds for any type of logic: classical, predicate, intuitionist, modal, linear logic. This observation is "trivial" because its effectively just a notational thing.

Alternative notations used to write inference rules, however, are interesting. One common form appearing in many computer-science settings is

$$X_1 \wedge X_2 \wedge \cdots \wedge X_n \rightarrow Y$$

The wedges obviously denote "conjunction", but the semantics of the X_k can be left wonderfully imprecise: are these boolean variables? Predicates? Or just terms of some sort? It doesn't much matter: the meaning of the wedge is say that all of these premises must be present and (perhaps) satisfied.

The LG notation for this is

$$X_1- \ \& \ X_2- \ \& \ \cdots \ \& \ X_n- \ \& \ Y+$$

One reason for writing the ampersand instead of a wedge is simply that the American keyboard does not have a wedge symbol on it, and LG dictionaries must be typed in by hand. The X_k are LG link types. The are not variables, they are not type-variables; they are types.

In tensor algebras, one would write

$$X_1^* \otimes X_2^* \otimes \cdots \otimes X_n^* \otimes Y$$

where the * denotes the (contra-variant) dual. In index notation, this would be written as

$$T_{\alpha\beta\cdots\mu}{}^{\nu}$$

In quantum mechanics, one uses the bra-ket notation:

$$|X_1\rangle \otimes |X_2\rangle \otimes \cdots \otimes |X_n\rangle \langle Y|$$

The tensor operator $\otimes$ is a kind-of conjunction, in that it states that all of the indicated terms must be present. It is also more: tensors can be assigned numeric values, and so $\otimes$ implies a certain kind of linearity on how tensors are composed from lower-rank tensors. Together with disjunction $\oplus$ and comultiplication, it forms a tensor algebra. There is a corresponding logic, called "linear logic" ("linear" because "linear algebra"). This is interesting because linear logic describes mutexes and semaphores in computing, as well as vending machines. Notable in the present context is that Link Grammar is a fragment of linear logic. Disconnecting connectors (the "type-raising combinator") appears to correspond to comultiplication.

One must be careful, though; the tensor forms can be beguilingly misleading. Tensor algebras are (dagger) symmetric, and thus have only one type constructor. In linguistics, there are two type constructors, which go to the left and the right, because the left-right distinction matters in linguistics. This is a source of confusion that hasn't been (in my mind) fully and clearly resolved. There's a further note on this at the end.

Connector Sexuality

In CCG, there are two "type raising combinators" $\backslash$ and $/$ because in linguistics, word order matters. Nouns appearing to the left of a verb are subjects; nouns appearing on the right are objects. Link Grammar accomplishes the left-right distinction with the $+$ and $-$ connector directions. This is, in general, sufficient for linearly-ordered sequences of words.

The rules for joining together LG connectors state that $+$ can only be attached to $-$. One can never attach $+$ to $+$ or $-$ to $-$. In this sense, the connection rules are heterosexual. This is also the conventional mechanism for lambda calculus, and of function calls: one can plug earlier outputs into new inputs. One can take a number 42 and plug it into $f(x)$ to get $f(42)$ but one cannot plug $f(x)$ into 42. Nor 42 into 42, for that matter. Function calls are also heterosexual (and almost always typed, except for simply-typed lambda calculus).

Mono-sexual connectors are those for which there is only one connector type. It can be denoted simply by *, or not at all (by just dropping the concept). In a monosexual system, all relations are necessarily homosexual, as there is only one sex.

Jigsaws can in general have monosexual connectors, or trisexual connectors, or other arbitrarily complex rules. If calling this "sex" seems odd, take a look at fungi, molds, mushrooms. Some have dozens of sexes, with complex mating rules!

Trisexuality

An example of trisexual connectors would be the set

$$\{Aa, Ab, Ac, Ba, Bb, Bc\}$$

with the connection rules that upper-case letters must match, while lower-case letters must be different. In this case, there are three sexes a, b, c instead of two $+-$, but the rules still demand heterosexuality between the connector "directions".

One can enliven the situation by introducing * as a direction wild-card. Thus, * can mate with *, or with any of the sexes a, b, c. So for example, Ab can attach to $A*$, or any of the other A's; just not to another Ab. Likewise $B*$ cannot attach to any of the A's, because the uppercase letters denote type, and you cannot mix these.

The fundamental need for connectors

We now come to perhaps the most subtle point of this. It's subtle because its blaringly, forehead-slappingly obvious. It's so obvious that, in fact, it will shoot right by, if you are not paying attention.

It is this: in almost all conventional, day-to-day usage of types, when someone says "this is a type", half the time, they really mean "this is a connector". Connectors are implicitly present almost everywhere; their use is rampant, and the concept of "direction" is never mentioned, because it is almost always obvious from context. One could say that type theory and computer programming suffer from "systemic hetero-sexism" or "normative heterosexuality".

Consider programming in C, C++ or Java. Three basic types are `int`, `float` and `string`. Duhh. Function calls have "signatures", *e.g.* `int func(int x)`. What is the number 42? Obviously, its an `int`, and obviously you can plug it into `int func(int x)`, so that `func(42)` is syntactically valid (in C, C++, Java) but `42(func)` is obviously syntactic nonsense. No one ever needs to explain this.

It would be strange and bizarre to explain that 42 is actually a "connector", having type `int` and direction "output". Likewise, in `func(int x)`, the `x` is actually a connector. Obviously, `x` has the type `int`, but it also has the direction of "input". There

is an implicit connector rule that states that connections can only be heterosexual. The rule is implicit because it's obvious: you can connect an output to an input, and that is it. You cannot connect two inputs, you cannot connect two outputs. Duhh. Any dummy knows this.

Now it is time to slap one's forehead. About half the time when someone says something is of type X, what they really mean is that something is a connector, and that connector has a type of X and a direction of either "input" or "output", which is always obvious from context. In software development, when people say "type", they often mean "connector".

Normative Heterosexuality

The reason for my belaboring this "normative heterosexuality" is that sometimes, it gets you into trouble. When CCG writes the inference rule

$$\frac{\alpha : X/Y \quad \beta : Y}{\alpha\beta : X} >$$

the types X and Y were implicitly mono-sexual. They were taken to have no directional information, and all left-right distinctions in the grammar emerged from the two type constructors $/$ and $\backslash$. Superficially, this seems all fine and correct, although ambiguous associative situations arise, which can be resolved by using parenthesis. Thus, associative expressions such as $X\backslash(Y/Z)$ and the algebra of CCG types is a non-associative algebra (the locations of the parenthesis matter).

In fact, the mono-sexed types, when used with the two combinators and with the parenthesis, provides a golden path to hell. This doesn't become apparent until one starts tripping over buggy expressions. The two example sentences contained three bugs, grand total. These bugs were not visible until the placeholders X and Y were reinterpreted as connectors (with types X and Y), and the previously implicit directional attachments were made explicitly visible with $+$ and $-$.

Type constructors vs. Sexuality

The definition of CCG involved seemingly mono-sexed types, and two type constructors $/$ and $\backslash$. The definition of LG involves heterosexual types, with connector directions $+$ and $-$ and a single type constructor &. This text has exposed the relationships between these two, but it leaves open a bigger question: what is the formal interplay between type constructors and sexuality? It seems that the one can be traded for the other, but the mechanics of this in a general setting are not clear.

Conclusion

The lesson for today: CCG is equivalent to LG. More or less. We glossed over or completely ignored many of the finer points of LG. No doubt, many important aspects of CCG were omitted as well. Yet, the basic jigsaw structure of CCG was exposed in the plainest way.

The meta-lesson for today: Jigsaws are fundamental for describing a vast class of mathematical and linguistic phenomena. Jigsaws have types (the types of the connectors) and the connectors have "sexuality" (usually heterosexual, for most applications).

The story does not end there; lets leave off with some hazy futuristic scifi: to infinity and beyond! Consider chemical bonds. Two atoms can bond to one-another, using ionic bonds, molecular bonds, hydrogen bonds and van der Waals bonds. In this sense, molecules are clearly jigsaw pieces, having connectors on them. The type+direction theory outlined so far is not quite sufficient to properly describe chemistry. But it does move in that direction. What more is needed to obtain a fully-accurate type-theoretic model of chemistry?

References

[1] Fei Xia and Martha Palmer., "Converting Dependency Structures to Phrase Structures", *HLT '01: Proceedings of the first international conference on Human language technology research*, 2001, pp. 1–5, URL https://aclanthology.org/H01-1014.pdf.

[2] Daniel Sleator and Davy Temperley., *Parsing English with a Link Grammar*, Tech. rep., Carnegie Mellon University Computer Science technical report CMU-CS-91-196, 1991, URL http://arxiv.org/pdf/cmp-lg/9508004.

[3] A. S. Troelstra and H Schwichtenberg, *Basic Proof Theory, Second Edition*, Cambrdige University Press, 2000.