Graphic Lambda Calculus and Natural Language Syntax

Linguistics 4403

Thomas Porter

July 1, 2022

1 Introduction

In this paper I present the advantages of extending the traditional tree-like semantic representations of function application, commonly used in formal semantics, with the graphic lambda calculus presented by Buliga (2013). I claim that traditional extensions to syntax trees, such as lines indicating coreference and arrows indicating movement correspond to components of the graphic lambda calculus.

2 Graphic Lambda Calculus

Traditional lambda calculus (with the order of application switched, following Barker) is given by the syntax:

$$e := x|\,\lambda\, x.e|e_1\, e_2$$

With x denoting an arbitrary variable. The semantics is given by the equivalence relations:

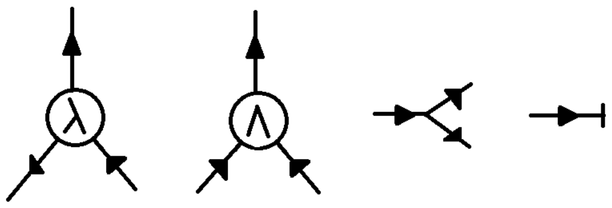
$$\lambda\, x.e[x] \equiv_\alpha \lambda\, y.e[y]$$

$$e_1(\lambda\, x.e_2) \equiv_\beta e_2[e_1/x]$$

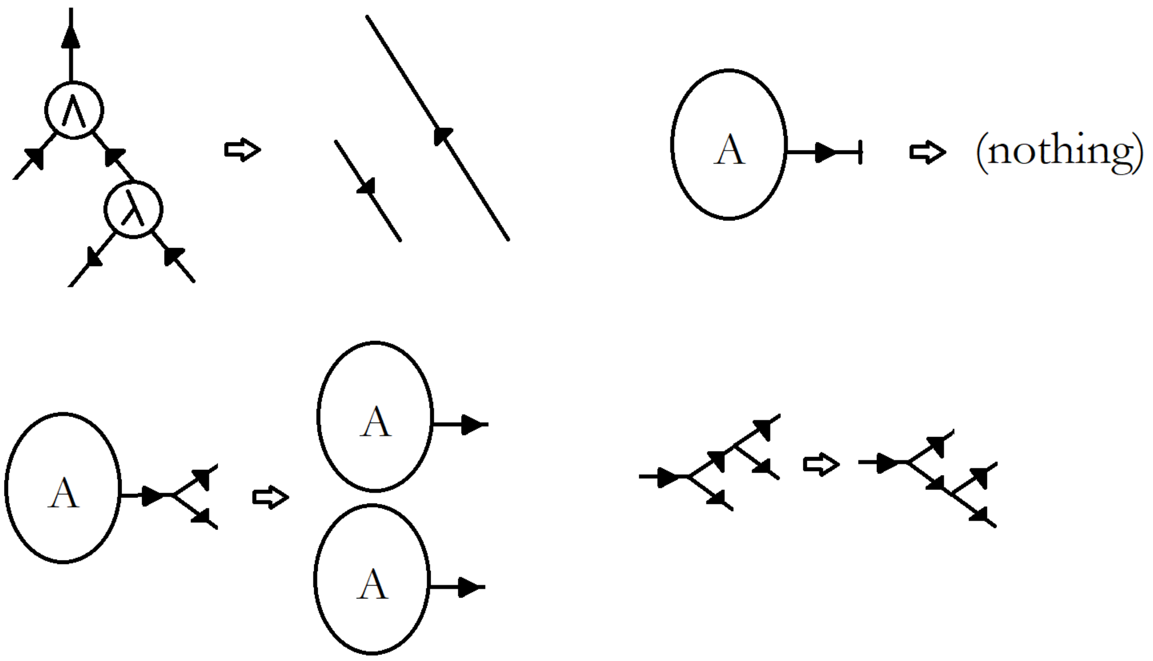
Where $e_2[e_1/x]$ denotes e_2 with all occurrences of x replaced with e_1. This system formalizes function definition, with $\lambda\, x.e[x]$ being the function that maps any x to $e[x]$, and function application, with $e_1 e_2$ being the argument e_1 fed in to the function e_2. It is also useful to consider a typed version of the lambda calculus, in which each term and variable is associated with a type, $\lambda\, x.e[x]$ has a function type (denoted $a \to b$, or $\underline{a}inlinguistics), and e_1\, e_2$ is only a legal application if e_2's argument type matches the type of e_1.

Buliga (2013) presents a graphic lambda calculus, combining the usual computational properties of the calculus with the graph-theoretic structure of the field of emergent algebras. Here I consider a graphic instantiation of the pure, basic lambda calculus, which has merits of its own, particularly related to variable collision, the parsing of lambda terms, and parsimony. In general, graphic lambda calculus reflects what is perhaps the true structure of the system, rather than compressing it into a linear sequence of symbols.

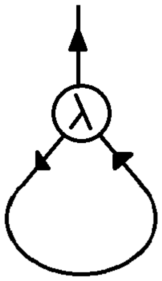
Terms in the graphic lambda calculus are directed graphs, the nodes of which are:



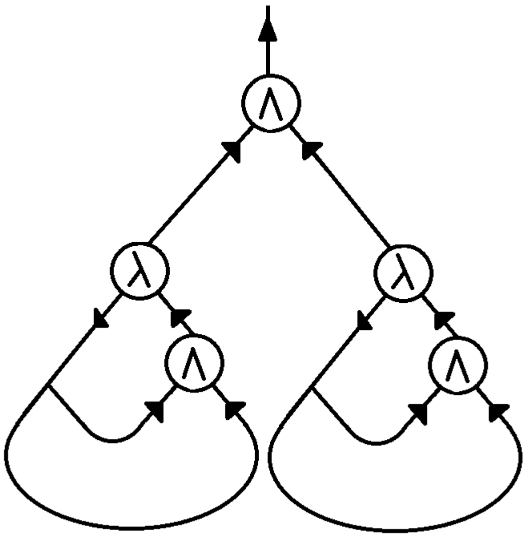
With the semantics given by the rewrite rules:



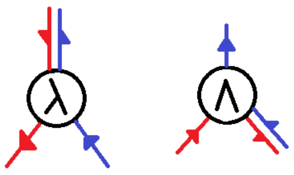
As well as various other rules that can be found in the original paper. Note that the first rule is analogous to the beta equivalence ($\equiv_\beta$) rule of the traditional lambda calculus, and that alpha equivalence is now automatic; variables are simply wires from the lambda node that binds them to all of their occurrences in the body of the abstraction. The correspondence between the traditional lambda calculus and the graphic lambda calculus is rather direct, but examples may be more helpful than a formal definition. Consider the identity function $I = \lambda\, x.x$:



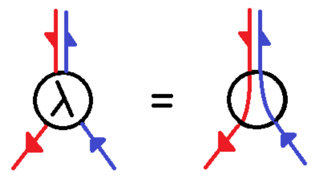
And the nonterminating $\Omega = (\lambda\, x.(xx))(\lambda\, x.(xx))$:

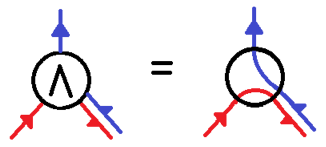


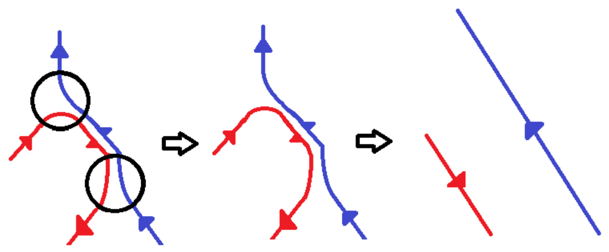
Note that types may be added to this system by "coloring" or in some way distinguishing the wires. This imposes new conditions on the syntax:



Note that the apparent double wire exiting the lambda node, and entering the merge node, is in fact a single wire that is meaningfully colored as the combination of two wire types oriented in opposite directions. In fact, we can peer deeper into these nodes, drawing them in such a way as to reduce beta reduction into mere wire shifting:







The given definition of the graphic lambda calculus technically includes various “nonsensical” graphs, such as a simple loop, which do not correspond to any lambda term, but for the purposes of this paper I will not consider them.

3 Merge

Of course, in this case the primary merit of graphic lambda calculus is its correspondence with natural language syntax. The primary instance of this is the correspondence between the merge operation in phrase structure and the merge operation in the lambda calculus. This correspondence is nothing new, and is in fact the mere interpretation of syntactic merge as function application. This is the foundational conceptual model of Lambek style grammar and categorical semantics in general.

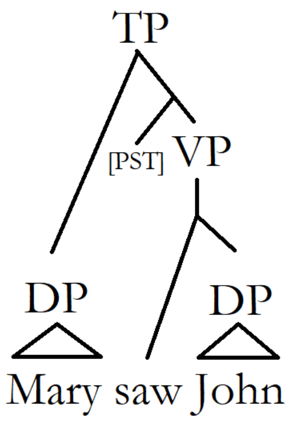
However, this correspondence requires stretching the lambda calculus is two ways. The first is that in Lambek grammar and in many semantic instances, it is necessary to express functions that take arguments from the left, and those that take arguments from the right. As presented, they can only take arguments from the left. This could be amended with a kind of "bidirectional lambda calculus," as follows:

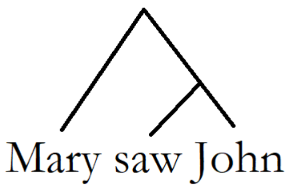
$$e := x |\, \lambda\, x.e | e.x\reflectbox{λ} | e_1\, e_2$$

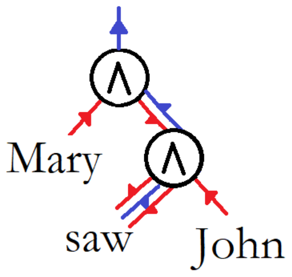
With the extra computational rule that $(e_1.x\reflectbox{$\lambda$})e_2 \equiv e_2(\lambda\, x.e_1)$. Now we may distinguish between types $A \to B$ (also written as $A \setminus B$) and $B \leftarrow A$ (also written as B/A). However, the untyped version of this new system is semantically ambiguous. Consider $(A.x\reflectbox{$\lambda$})(\lambda\, x.B)$, for arbitrary constant expressions (not containing x) A and B. This could evaluate to either A or B, depending on which side of the merge is the function, and which is the argument. This in turn could be sidestepped with two notions of merge, one for the left and one for the right, but in this case the semantics would identical to that of the unidirectional lambda calculus, with an meaningless degree of freedom in how expressions are written. It seems that the problem of inferring which side is the function and which is the argument in the single merge case is tantamount to deciding which semantic merge is associated with the given syntactic merge in the double merge case.

The other modification is the very common practice of extending the lambda calculus with arbitrary terms, which we may think of as lexical terms in this context. Each such term has its own type and computational content, may be applied or applied to. This has the effect of allowing terms of the type $B \leftarrow A$ without the need to introduce bidirectional lambda abstraction, if only lexical items contain types of that form. Indeed, Barker finds unidirectional lambda abstractions sufficient when paired with a lexicon of bidirectionally typed terms. Note that this limits quantifier movement, for example, to be towards the beginning of the sentence. It seems plausible that this has cross-linguistic justification, but also that this may not be sufficient for absolutely all purposes.

The advantage of graphic lambda calculus is that when merge is the only operation, the semantic graphs mirror the syntactic parse trees in the simply compositional case.







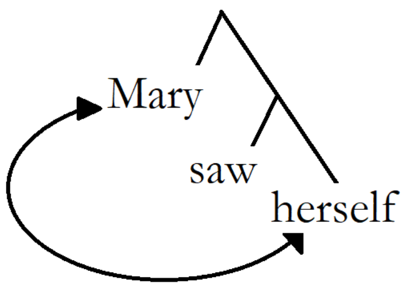
On the left is the syntax tree, in the middle is a simplified version that only conveys constituency data, and on the right is the graphic lambda calculus semantic representation. Note that this includes the bidirectional merge.

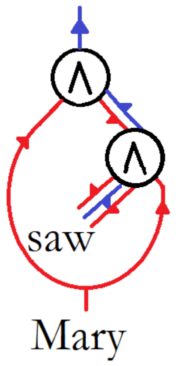
4 Abstraction

The more interesting cases are those in which the lambda calculus term actually includes (non-constant) lambda abstractions, or equivalently, when the graphic lambda calculus term is not a tree. We will examine several cases in which this happens, focusing on the analogy between the graphic lambda calculus and syntactic diagrams of scope and movement.

4.1 Coreference

“Mary saw herself.”



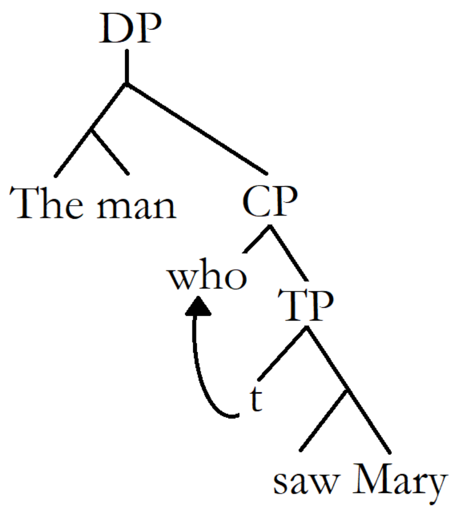


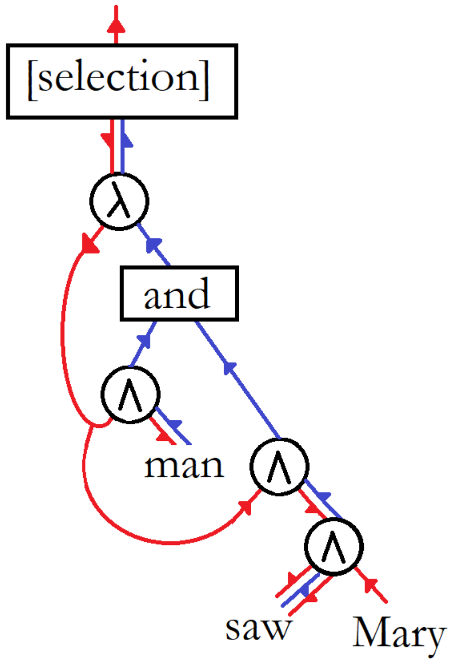
On the left is the corresponding syntax tree with a wire indicating coreference, and on the right is a semantic representation. The wires connecting coreferential expressions superposed on the syntax tree correspond to a branching wire from the same origin in the semantic representation. Red here represents the type e of entities and blue represents the type t of truth values.

The explanation of this correspondence lies in the fact that while any linear or symbol presentation of function composition is naturally structured as a tree, with each function expression requiring arguments, the semantics of function composition correspond to directed, acyclic graphs. This is because the output of a function may be used as the input to multiple other functions, causing an undirected cycle in the graph. In other words, two constituents are coreferential. The introduction of the diagonal node (a branching wire) into the graphical system allows the expression of acyclic graphs of function composition, corresponding to coreference.

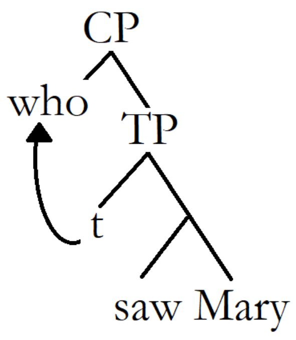
4.2 Relative Constructions

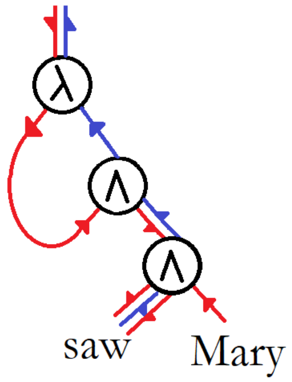
"The man who saw Mary."



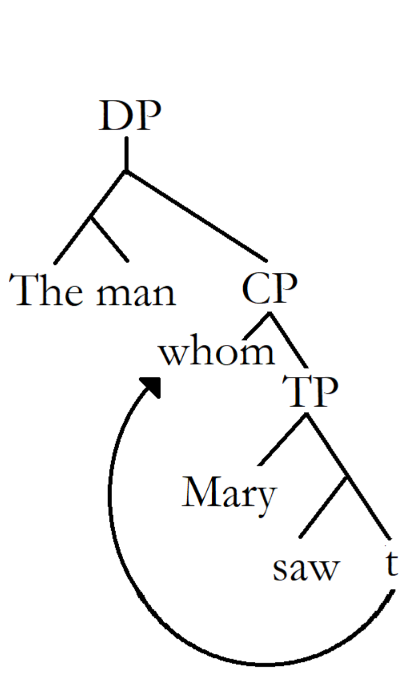


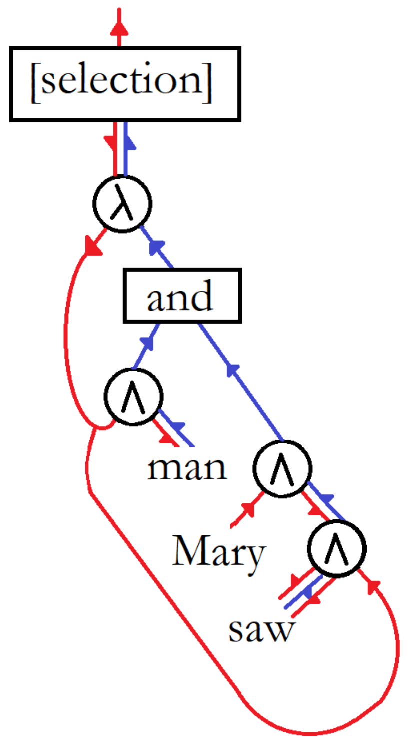
The structure is clearer when just the relative clause, of unary proposition type $(e \to t)$, is represented.





“The man whom Mary saw.”

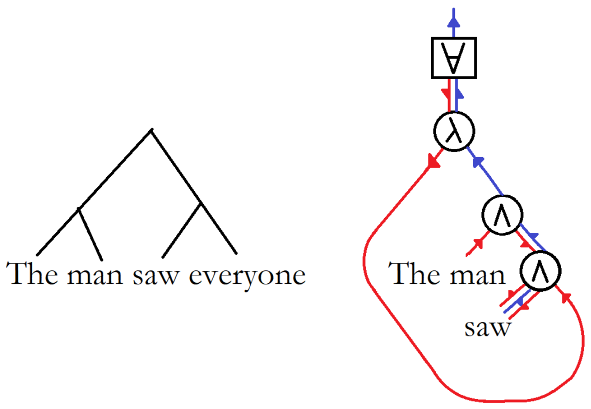




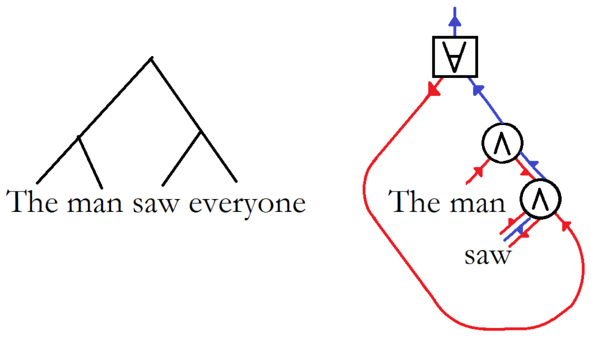
In the movement interpretation of relative constructions, the movement arrow of the relative wh-element corresponds to the bound variable wire of a lambda abstraction. This corresponds to the traditional interpretation of relatives as creating lambda abstractions, with the abstraction head at the movement target and the abstraction variable at the trace. The use of the graphical lambda calculus does not posit any further theoretical predictions. Instead, it allows the expression of such structures in a more natural way, visibly representing the relationships across the syntax tree, and without requiring the choice of variable names. The principle convenience comes from the practice of enhancing the syntactic representation by drawing arrows for movement of relatives, which now is in formal correspondence with the enhanced semantic representation.

4.3 Quantifier Raising

"The man saw everyone."

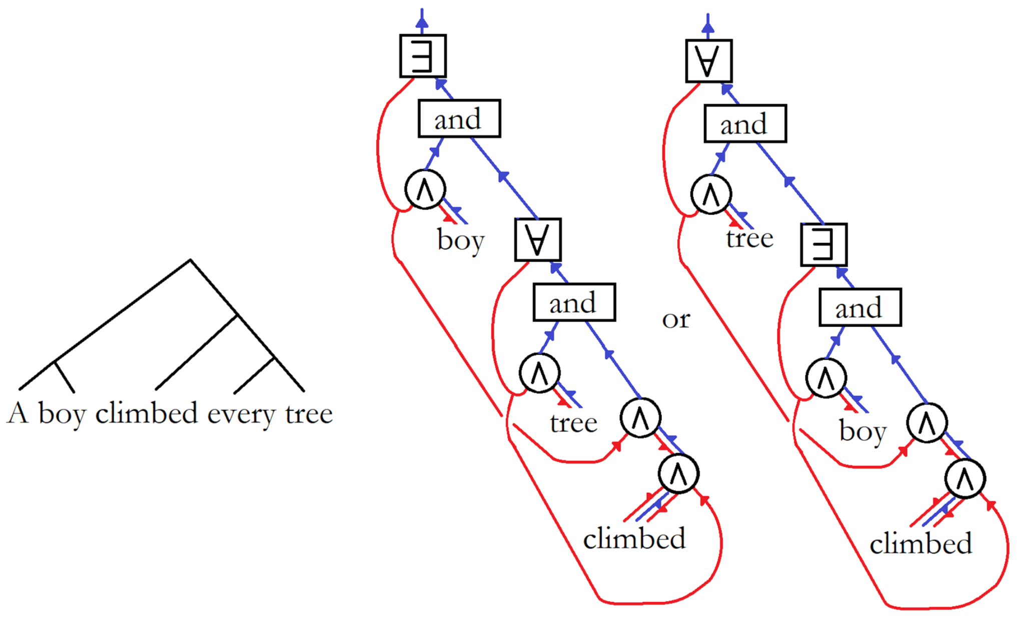


In the movement interpretation of quantifier raising, the movement arrow of the quantifier corresponds to the bound variable wire of a lambda abstraction. Note that this semantic representation assumes quantifiers are of type $t/(e \setminus t)$, although they might equally well be more naturally represented as an alternative to a lambda node, rather than one that requires a lambda node:



This is the graphic instantiation of Quine-Bourbaki notation for quantifier bindings (Button & Walsh, p. 14), which is similar in spirit to De-Bruijn notation, a linear predecessor to the full graphic lambda calculus.

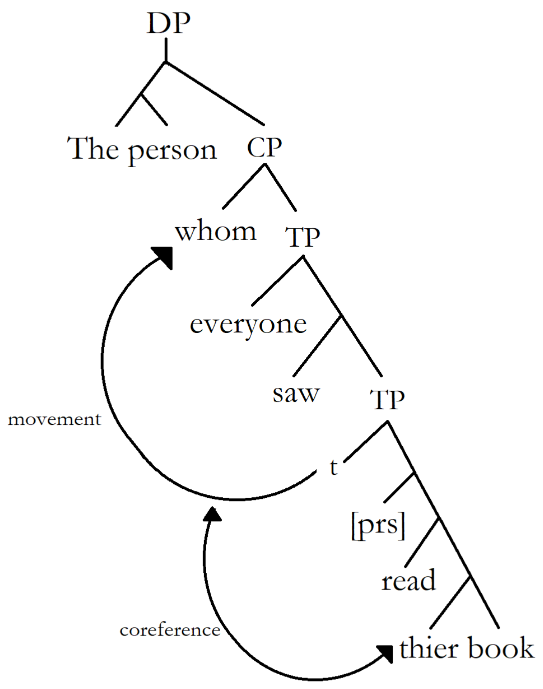
“A boy climbed every tree.”

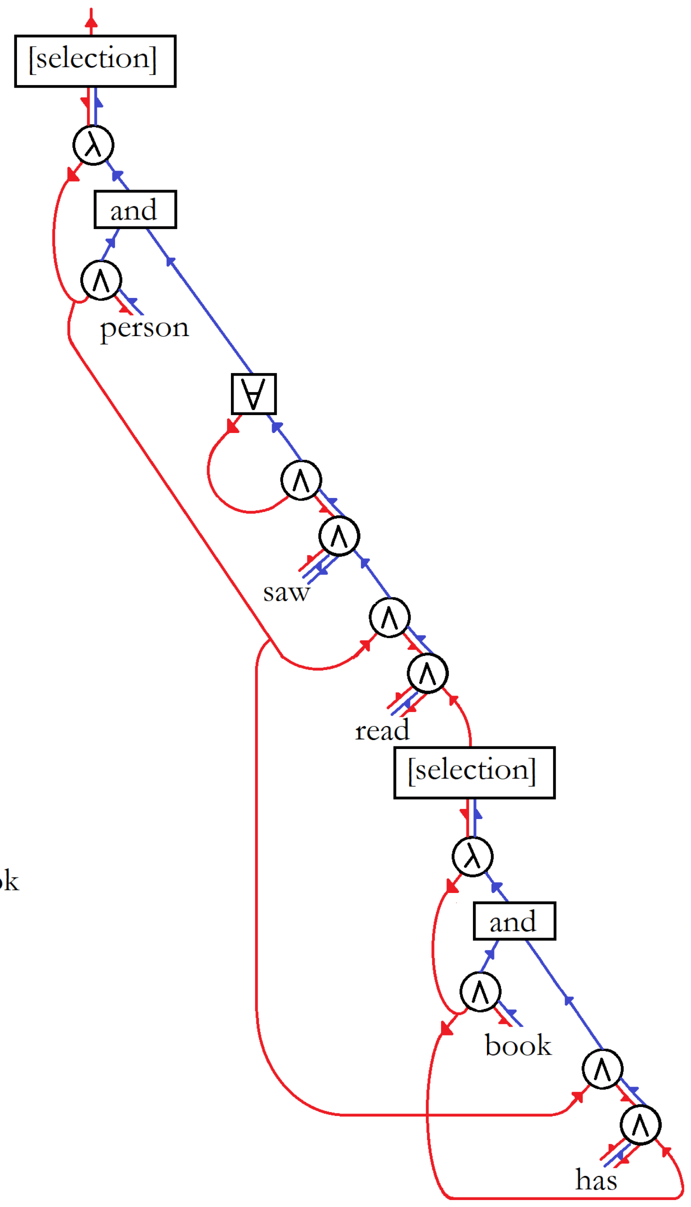


The theory of quantifier raising, often presented using lambda abstractions as in Barker (2020), makes various predictions about the allowed locations to which quantifiers can be raised from where, but again the graphic framework does not enforce any predictions. Both cases of the ambiguous sentence above can be represented, as well as various ungrammatical structures. The framework simply elucidates existing models.

4.4 Interaction

“The person whom everyone saw read (PRS) their book”





The above is an example of a relative clause with quantifier raising and coreference. Note the compact yet informative structure of the semantic representation. It is notable that although the three phenomena listed here all have instantiations in the graphic lambda calculus, they appear to be largely independent and do not appear to conflict with each other.

5 Scope

Although the graphic lambda calculus can be used to express semantic content independently of making predictions, it may be useful to interpret the theory of scope in the graphical case. The question of how the C-Command relation influences scoping judgements for bound variables is discussed in Moulton and Han (2018). They find that quantifier, quantified pairs in a C-Command relation enforce constraints beyond what would otherwise be the case.

Under the strict interpretation of syntactic merge as corresponding to semantic merge, which is not always the case, A C-commands B iff the denotation of A takes the denotation of a phrase containing B as an argument, or if the denotation of A is the argument to the denotation of a phrase containing B. However, in the case of a lambda abstraction node, the correspondence of merges is broken. Instead, what manifests in the semantics as a lambda node with a subgraph beneath it is a lambda-triggering lexical item that has been moved into the head of a phrase, e.g. CP. This suggests that the C-command domain of the quantifier corresponds roughly to the subgraph beneath the lambda node. This is the principle mode of semantic scope, but perhaps may not be the case in the case of a co-varying pronoun, since there are two lambda phenomena occurring: quantification and coreference. Perhaps the difference between C-Commanding scope and not C-Commanding scope may be explained by the difference between the principle subgraph beneath the lambda node, and the rest of the semantic graph.

6 Conclusion

The scoping phenomena of coreference, wh-element movement, and quantification require the extension of general function trees with lambda abstractions. I here have shown the graphical interpretation of this extension, and argued that the graphic lambda calculus provides the more natural representation of these scoping phenomena, particularly in their geometric analogy to syntactic markings beyond the phrase structure syntax tree. I also discuss the relation to C-Commanding scope, and the asymmetry of introducing the lambda calculus into a pregroup (Lambek) grammar, in that this only allows movement towards the front of sentences. A sketch of the more symmetrical, “bidirectional” lambda calculus is given.

7 References

Barker, Chris. 2020. The logic of Quantifier Raising. *Semantics and Pragmatics* **13** (2020)

Buliga, Marius. 2013. Graphic lambda calculus. *Complex Systems* **22, 4** (2013), 311-360

Button, Tim and Sean Walsh. 2018. Philosophy and Model Theory. *Oxford University Press*

Moulton, Keir and Chung-hye Han. 2018. C-Command vs. Scope: An Experimental Assessment of Bound Variable Pronouns. *Language* **94(1)**, 191-219