NLQ: a modular type-logical grammar for quantifier movement, scope islands, and more

Pepijn Kokke

A thesis submitted in fulfilment of the requirements
for the degree of Master of Science

Under the supervision of prof. dr. Michael Moortgat
and dr. Wouter Swierstra

February 17, 2016

I would like to thank Michael Moortgat for introducing me to the wonderful field of type-logical grammar, and for supervising me throughout this thesis.

I would like to thank Wouter Swierstra for introducing me to dependent type theory, which has become so central to my understanding of logic and mathematics, and to Agda, my favourite programming language.

And finally, I would like to thank Chris Barker, Oleg Kiseljov, Jirka Maršík and Richard Moot for the incredibly useful lectures and discussions on NL_λ, scope, continuations, and monadic semantics at ESSLLI 2015.

Contents

1 Introduction

In this thesis, I will discuss the grammar logic NLQ, an extension of the non-associative Lambek calculus, which is capable of analysing quantifier movement, scope islands, infixation and extraction.

What I hope to do in this thesis is to *extend* and *solidify* the logical vocabulary with which such linguistic analyses can be made. In this, I will use the following guiding principles:

- We are constructing a *grammar* logic. Therefore, we only want features in our logic for which we can demonstrate a motivating example from natural language.
- We are constructing a grammar *logic*. Therefore, we will only accept extensions to our logic if we can show that they preserve our most important properties: we want our logic to be reflexive and transitive, and want a procedure for proof search that is both *decidable* and *complete*.

I am under no impression that the extensions I am proposing will be the be-all and end-all of logical grammar, so another important point in this thesis will be *modularity*. It is incredibly important to formulate extensions in a modular manner, so that other logical grammarians are free to mix and match extensions without having to worry about unforeseen interactions. There are two key techniques for this: (1) we use display calculus to get a general procedure for cut-elimination (section 2); and (2) we associate each syntactic extension with its own set of connectives (or *modality*) and make sure that the inference rules in that extension only apply in the presence of these connectives (section 4.3).

Another key point will be *unique normal-forms*—in our proof search procedure, we only want to find a single proof for each interpretation that a sentence has. In our calculus, we will achieve this using *focusing* (section 2.4).

The last key point in this thesis will be *verification*. It is far too easy to make mistakes when writing down logical proofs in a pen-and-paper style, or when manually typesetting them in LaTeX. Therefore, most of the claims I make in this thesis will be backed up by a pair of verified implementations of the full version of NLQ (i.e. the version using all discussed extensions). These verifications can be found in appendices A and B and on GitHub.

In appendix A, we discuss a formalisation in Agda (Norell, 2009). We implement the grammar logic, prove some key properties, and give a formal semantics in the form of a translation from proofs in NLQ to Agda terms.

In appendix B, we discuss a formalisation in Haskell (Marlow, 2012) using the singletons library Eisenberg and Weirich (2012). In the full version, we implement the grammar logic, implement proof search, and give a formal semantics in the form of a translation into a subset of Haskell which includes meaning postulates. However, because the implementations of the grammar logic are nearly identical, we restrict our discussion of the Haskell version to the interface provided by the library, and how to write your own lexicon and example sentences.

Starting in section 1.1, we will give a brief introduction to type-logical grammar in general, and to what I consider to be the base type-logical grammar: the non-associative Lambek calculus (NL) paired with a simple semantic lambda calculus ($\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$). In section 2, we will discuss the display calculus formulation

of NL, and motivate our usage of display calculus. Then, in sections 3 and 4, we will discuss several extension to the base type-logical grammar.

1.1 What is type-logical grammar?

Before we address the question of what type-logical grammar is, let us try and get an idea of what problem it is trying to solve. Have a look at the abstract pipeline for natural language understanding (NLU) in figure 1.

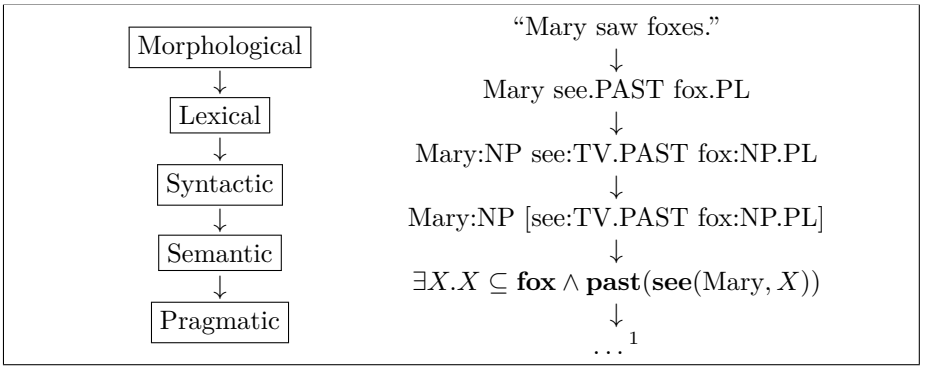
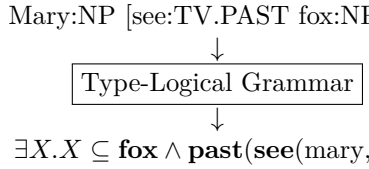


Figure 1: An abstract pipeline for natural language understanding.

To the left of the figure, you see the various phases or functions commonly associated with an NLU-pipeline. To the right, you see the inputs and outputs of these functions. For instance, the morphological function will take an unanalysed sentence, and return a sentence which is lemmatised. This entails that all morphemes are made explicit—for instance, in the case of the example in figure 1, the previously "implicit" morphemes for past tense and plurality are added.

There is some disagreement on the exact role of type-logical grammars in this pipeline. Ideally, type-logical grammars would play the role of both the syntactic and the semantic function. However, the current state of affairs in research is that often only the semantic function is truly considered[2]. This makes sense from a research perspective: we can refer to the huge body of work on generative grammar to inform our choice for sentence structure, and focus on assigning the right meaning to these structures. This is also the approach we will also take in this thesis—that is, we consider type-logical grammars to be the function:



[1] As the pragmatic function is all about integrating context (be it textual or environmental) into the meaning, it would not make sense to list something here.

[2] This statement is not true for *associative* type-logical grammars, which fundamentally reject the tree structure of language—that is, they assume that the meaning of a sentence depends solely on the linear order of words, and not on some hidden tree structure.

That is, it is a function which, given some structured and typed input which represents the syntactic structure of a sentence, returns the meaning(s) associated with that sentence.

Given the presence of the phrase "structured and typed", we may already suspect that type theory offers a fitting solution to this problem. And indeed, under the guise of type-logical grammar, it does. A type-logical grammar generally consists of three things:

(1) a syntactic calculus, set up in such a fashion that only grammatical sentences are well-typed, and for which an efficiently decidable procedure for proof-search exists;

(2) a semantic calculus, used to represent the meanings of words and sentences; and

(3) a translation from the syntactic to the semantic calculus.

We interpret the part-of-speech tags in our input (NP, TV, etc.) as types in the syntactic calculus, and combine these with the desired type for the tree—usually S for 'sentence'—to form an input sequent. We then search for a proof of that sequent in the syntactic calculus, and translate it to a term in the semantic calculus. Once there, we interpret the morphemes (e.g. lemmas, `PAST`, `PL`, etc.) as terms in the semantic calculus.

In section 1.2, we will have a look at the base type-logical grammar, and give some examples of the process of deriving sentence meaning.

1.2 A simple type-logical grammar

The simplest type-logical grammar that comes to mind—drawing heavily from Montague grammar and categorial grammar—is composed of the simply-typed lambda calculus with atomic types $\mathbf{e}$ and $\mathbf{t}$ ($\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$) as a semantic calculus, and the non-associative Lambek calculus (NL; Lambek, 1961) as a syntactic calculus.

The usual natural deduction formulation of $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$ can to be seen in figure 2. It is a simple lambda calculus, with atomic types $\mathbf{e}$ ('entity') and $\mathbf{t}$ ('truth-value'). In addition, we usually assume that any logical operator or word-meanings we need is defined as a constant of the appropriate type. For instance, $\forall$ is a constant of type $(\mathbf{e} \to \mathbf{t}) \to \mathbf{t}$, and 'john' is a constant of type $\mathbf{e}$. Note that we will sometimes write logical operators in their usual notation, e.g. $M \wedge N$ or $\forall x.M$, but this should be taken as syntactic sugar, in the case of our examples rewriting to $((\wedge\ M)\ N)$ and $\forall\ (\lambda x.M)$, respectively. Additionally, we will occasionally write e.g. $\mathbf{eet}$ instead of $\mathbf{e} \to \mathbf{e} \to \mathbf{t}$, or $(\mathbf{et})\mathbf{t}$ instead of $(\mathbf{e} \to \mathbf{t}) \to \mathbf{t}$, using adjacency to mean implication.

Using this calculus as a semantics function directly would over-generate, e.g. for the sequent $\{\text{john} : \mathbf{e}, \text{likes} : \mathbf{e} \to \mathbf{e} \to \mathbf{t}, \text{mary} : \mathbf{e}\} \vdash \mathbf{t}$ we can derive ((likes john) mary), ((likes mary) john), ((likes mary) mary) and ((likes john) john). The reason for this is, of course, that the set structure used in this formulation is much too expressive for natural language grammar.

If we want more control over the structure of our terms, a good first step is to move to a purely syntactic formulation, where all the structural properties are made explicit in the calculus itself; this has been done in figure 3. We have replaced the set by a (possibly empty) binary tree, spanned by the structural

$$
\begin{array}{rl}
\text{Atom } \alpha & ::= \mathbf{e} \mid \mathbf{t} \\
\text{Type } A, B & ::= \alpha \mid A \to B \\
\text{Term } M, N & ::= x \mid C \mid \lambda x.M \mid (M\ N) \\
\text{Constant } C & ::= \forall \mid \exists \mid \neg \mid \supset \mid \wedge \mid \vee \mid \ldots \\
\text{Environment } \Gamma & \text{set of typing assumptions of the form '} M : A\text{'}
\end{array}
$$

$$
\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A}\ \text{Ax}
$$

$$
\frac{\Gamma, x : A \vdash M : B}{\Gamma \vdash \lambda x.M : A \to B} \to\text{I} \qquad \frac{\Gamma \vdash M : A \to B \qquad \Gamma \vdash N : A}{\Gamma \vdash (M\ N) : B} \to\text{E}
$$

Figure 2: $\lambda^{\to}_{\{\mathbf{e},\mathbf{t}\}}$, a simple semantic calculus.

product '$\bullet$'. We have also included a number of new structural rules, which implement the structure of a set: $\varnothing$E and $\varnothing$I allow us to have an empty antecedent; contraction and weakening tell us that we can use formulas multiple times or not at all; and with commutativity and associativity we can change the order of the formulas any way we like.

Note that, in order to define these structural rules, we had to define the notion of a 'context'—a structure with *exactly one* hole in it—and a plugging function '$\cdot\,[\,\cdot\,]$'—a function which inserts a structure into that hole. The reason for this is that we have to be able to apply commutativity and associativity *anywhere* in the structure to be able to freely change the order (and bracketing).[3]

It is not hard to convince yourself that the implicit and explicit versions of $\lambda^{\to}_{\{\mathbf{e},\mathbf{t}\}}$ are equivalent—though we will refrain from giving the full proof here. Because of this equivalence, we can use the term language from figure 2 for the explicit version of $\lambda^{\to}_{\{\mathbf{e},\mathbf{t}\}}$. The term labelling of the logical rules is exactly the same. The structural rules only manipulate structures, and therefore do not change the terms. The only exception to this is contraction, for which the term labelling is as follows:

$$
\frac{\Sigma[y : A \bullet z : A] \vdash M : B}{\Sigma[x : A] \vdash M[x/y][x/z] : B}\ \text{Cont.}
$$

Contraction takes a term with two variables of the same type, and contracts

[3]The contexts are not strictly necessary for $\varnothing$E, contraction and weakening, since we can already move any formula anywhere we want, but they make the proof system much more usable and greatly decrease the length of proofs that need to use any of these structural rules.

$$
\begin{array}{rl}
\text{Atom } \alpha & ::= \mathbf{e} \mid \mathbf{t} \\
\text{Type } A, B & ::= \alpha \mid A \to B \\
\text{Structure } \Gamma, \Delta, \Pi & ::= A \mid \varnothing \mid \Gamma \bullet \Delta \\
\text{Context } \Sigma & ::= \Box \mid \Sigma \bullet \Delta \mid \Gamma \bullet \Sigma
\end{array}
\qquad
\begin{array}{c}
\Box[\Gamma] \mapsto \Gamma \\
(\Sigma \bullet \Delta)[\Gamma] \mapsto (\Sigma[\Gamma] \bullet \Delta) \\
(\Delta \bullet \Sigma)[\Gamma] \mapsto (\Delta \bullet \Sigma[\Gamma])
\end{array}
$$

$$\frac{}{A \vdash A}\ \text{Ax}$$

$$\frac{\Gamma \bullet A \vdash B}{\Gamma \vdash A \to B} \to\text{I} \qquad \frac{\Gamma \vdash A \to B \qquad \Delta \vdash A}{\Gamma \bullet \Delta \vdash B} \to\text{E}$$

$$\frac{\Sigma[\Gamma \bullet \varnothing] \vdash B}{\Sigma[\Gamma] \vdash B} \varnothing\text{E} \qquad \frac{\Sigma[\Gamma] \vdash B}{\Sigma[\Gamma \bullet \varnothing] \vdash B} \varnothing\text{I}$$

$$\frac{\Sigma[A \bullet A] \vdash B}{\Sigma[A] \vdash B}\ \text{Cont.} \qquad \frac{\Sigma[\Gamma] \vdash B}{\Sigma[\Gamma \bullet A] \vdash B}\ \text{Weak.}$$

$$\frac{\Sigma[\Delta \bullet \Gamma] \vdash B}{\Sigma[\Gamma \bullet \Delta] \vdash B}\ \text{Comm.} \qquad \frac{\Sigma[(\Gamma \bullet \Delta) \bullet \Pi] \vdash B}{\Sigma[\Gamma \bullet (\Delta \bullet \Pi)] \vdash B}\ \text{Ass.}$$

Figure 3: $\lambda^{\to}_{\{\mathbf{e},\mathbf{t}\}}$, with explicit structural rules.

them using substitution, which is defined as usual:

$$
\begin{array}{ll}
x & [N/y] \mapsto \begin{cases} N, & \text{if } x = y \\ x, & \text{otherwise} \end{cases} \\
C & [N/y] \mapsto C \\
(\lambda x.M) & [N/y] \mapsto \begin{cases} \lambda x.M[N/y], & \text{if } x = y \\ \lambda x.M, & \text{otherwise} \end{cases} \\
(M\ M') & [N/y] \mapsto (M[N/y]\ M'[N/y])
\end{array}
$$

Using our explicit semantic calculus, we can construct our syntactic calculus in three simple steps:

1. we drop *all* structural rules;
2. since the implication '$\to$' can now only take arguments directly from the left, we add a second implication '$\leftarrow$' which can only take arguments from the right—by convention, implications in this system are written as '$\backslash$' and '$/$' (pronounced "under" and "over") with the argument type written *under* the slash;
3. we replace the atomic semantic types $\mathbf{e}$ and $\mathbf{t}$ by atomic syntactic types, reminiscent of part-of-speech tags—in this case, we will use S ('sentence'), NP ('noun phrase'), N ('noun'), PP ('prepositional phrase') and INF ('infinitive');

The resulting system can be seen in figure 4, defined along with some definitions for common part-of-speech tags, i.e. A ('adjective'), IV ('intransitive verb') and TV ('transitive verb').

$$\begin{array}{rll} \text{Atom } \alpha & ::= \text{S} \mid \text{N} \mid \text{NP} \mid \text{INF} & \text{A} \quad ::= \text{N} \,/\, \text{N} \\ \text{Type } A, B & ::= \alpha \mid A \setminus B \mid B \,/\, A & \text{IV} \quad ::= \text{NP} \setminus \text{S} \\ \text{Structure } \Gamma, \Delta & ::= A \mid \Gamma \bullet \Delta & \text{TV} \quad ::= \text{IV} \,/\, \text{NP} \end{array}$$

$$\frac{}{A \vdash A}\,\text{Ax}$$

$$\frac{A \bullet \Gamma \vdash B}{\Gamma \vdash A \setminus B}\setminus\text{I} \qquad \frac{\Gamma \vdash A \qquad \Delta \vdash A \setminus B}{\Gamma \bullet \Delta \vdash B}\setminus\text{E}$$

$$\frac{\Gamma \bullet A \vdash B}{\Gamma \vdash B \,/\, A}\,/\text{I} \qquad \frac{\Gamma \vdash B \,/\, A \qquad \Delta \vdash A}{\Gamma \bullet \Delta \vdash B}\,/\text{E}$$

Figure 4: NL (Lambek, 1961) in natural deduction style.

$$\begin{array}{ll} \text{S}^* & \mapsto \mathbf{t} \\ \text{N}^* & \mapsto \mathbf{e} \to \mathbf{t} \\ \text{NP}^* & \mapsto \mathbf{e} \\ \text{INF}^* & \mapsto \mathbf{e} \to \mathbf{t} \\ (A \setminus B)^* & \mapsto A^* \to B^* \\ (B \,/\, A)^* & \mapsto A^* \to B^* \end{array}$$

$$\frac{}{x : A \vdash x : A}\,\text{Ax}$$

$$\frac{x : A \bullet \Gamma \vdash M : B}{\Gamma \vdash \lambda x.M : A \setminus B}\setminus\text{I}$$

$$\frac{\Gamma \vdash N : A \qquad \Delta \vdash M : A \setminus B}{\Gamma \bullet \Delta \vdash (M\ N) : B}\setminus\text{E}$$

$$\frac{\Gamma \bullet x : A \vdash M : B}{\Gamma \vdash \lambda x.M : B \,/\, A}\,/\text{I}$$

$$\frac{\Gamma \vdash M : B \,/\, A \qquad \Delta \vdash N : A}{\Gamma \bullet \Delta \vdash (M\ N) : B}\,/\text{E}$$

Figure 5: NL (figure 4) with term labelling in $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$ (figure 2).

Dropping *all* structural rules may seem unnecessary, but there is a good motivation for each rule. For example, in the presence of commutativity, there is no way to distinguish between "Mary walks" and "walks Mary"; under weakening, we can add any word anywhere in a grammatical sentence, and the sentence will remain grammatical— e.g. "Mary banana walks"; and with contraction, we can remove consecutive words with the same type—which means that "John read

a fantastic blue book" could be taken to mean the same thing as "John read a blue book".

With respect to associativity, Lambek (1961, p. 167) mentions that "the most natural assignments of types to English words [would] admit many pseudo-sentences as grammatical, e.g.

(*) *John is poor sad. John likes poor him. Who works and John rests?*

More examples, including specific derivations, of ungrammatical sentences that would be admitted in the presence of associativity and the empty structure can be found in Moot and Retoré (2012, p. 33, 105-106).

Note that we use the product-free version of NL. The reason for this is that we have no use for the product in this thesis. Should you need the product, however, it is very easily added:

$$\frac{\Gamma \vdash A \qquad \Delta \vdash B}{\Gamma \bullet \Delta \vdash A \otimes B}\,\mathrm{L}\otimes \qquad \frac{\Gamma \vdash A \otimes B \qquad \Sigma[A \bullet B] \vdash C}{\Sigma[\Gamma] \vdash C}\,\mathrm{R}\otimes$$

The last component we need for our simple type-logical grammar is a translation from our syntactic calculus to our semantic calculus, which consists of:

(1) a function $(\cdot)^*$, translating the types in NL to types in $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$; and

(2) a set of rewrite rules, that rewrite proofs in NL to proofs in $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$.

However, in the interest of brevity, we will often give this second translation directly as a term labelling. For instance, in figure 5, we give the translation on terms by directly labelling the rules of the syntactic calculus with semantic terms. Because there is a one-to-one correspondence between lambda terms and proofs, this is perfectly unambiguous.

Note that we have chosen the particular translation for atomic types in figure 5 because it aligns well with the remainder of this thesis. However, there are different ways to define this translation—most notably, Montague's (1973) worst-case generalisation for NPs, which interprets them as having the type $(\mathbf{et})\mathbf{t}$.

Now that we have a full type-logical grammar, let's give an example analysis of the sentence "Mary likes Bill". We assume the morphological, lexical and syntactic phases have been taken care of, which leaves us with the following endsequent:

$$\text{mary} : \text{NP} \bullet (\text{likes} : \text{TV} \bullet \text{bill} : \text{NP}) \vdash ? : \text{S}$$

Fortunately, proof search is decidable for this system, so we can simply search the space of all possible proofs of this sequent. As it turns out, the only proof is:

$$\dfrac{\dfrac{}{\text{mary} : \text{NP} \vdash \text{mary} : \text{NP}}\,\text{Ax} \qquad \dfrac{\dfrac{}{\text{likes} : \text{TV} \vdash \text{likes} : (\text{NP} \setminus \text{S}) \,/\, \text{NP}}\,\text{Ax} \qquad \dfrac{}{\text{bill} : \text{NP} \vdash \text{bill} : \text{NP}}\,\text{Ax}}{\text{likes} : \text{TV} \bullet \text{bill} : \text{NP} \vdash (\text{likes bill}) : \text{NP} \setminus \text{S}}\,/\text{E}}{\text{mary} : \text{NP} \bullet (\text{likes} : \text{TV} \bullet \text{bill} : \text{NP}) \vdash ((\text{likes bill}) \text{ mary}) : \text{S}}\,\setminus\text{E}$$

And so, by searching for a proof in our syntactic calculus (bottom-up) and then adding in the term labelling (top-down) we derive a function-argument structure for our sentence. Usually, we include another step in this process,

where we insert the lexical definitions for the words. For the above example, these are:[4]

$$\begin{aligned} \text{mary} &= \mathbf{mary} \\ \text{john} &= \mathbf{john} \\ \text{likes} &= \lambda y.\lambda x.\mathbf{like}(x, y) \end{aligned}$$

After inserting these definitions, and β-reducing, we get:

$$\mathbf{like}(\mathbf{john}, \mathbf{mary})$$

Because we are usually only interested in the resulting function-argument structure and the associated semantics, for the remainder of this thesis we will summarise the above translations as follows:

$$\dfrac{\dfrac{}{\text{NP} \vdash \text{NP}}\,\text{Ax} \qquad \dfrac{\dfrac{}{\text{TV} \vdash (\text{NP} \setminus \text{S}) \,/\, \text{NP}}\,\text{Ax} \qquad \dfrac{}{\text{NP} \vdash \text{NP}}\,\text{Ax}}{\text{TV} \bullet \text{NP} \vdash (\text{NP} \setminus \text{S}}\,/\text{E}}{\text{NP} \bullet (\text{TV} \bullet \text{NP}) \vdash \text{S}}\,\setminus\text{E}$$

$$\downarrow$$

$$((\text{likes bill})\ \text{mary})$$

$$\downarrow$$

$$\mathbf{like}(\mathbf{john}, \mathbf{mary})$$

1.3 Sequent calculus and proof search

In the previous section, we glossed over the issue of proof search. This is problematic, because the natural deduction formulation of the syntactic calculus we presented in figure 4 is not especially suited to proof search. Lambek originally developed a sequent calculus for NL, which *does* have a practical procedure for proof search. In figure 6 we present the product-free version of Lambek's (1961) sequent calculus.

One important property of sequent calculus is the *sub-formula* property—the property that a derivation of a sequent uses only proper sub-formulas of the formulas in that sequent. As a direct consequence of this property, we generally get an algorithm for proof search which is both easy to implement, and complete. This algorithm is backward-chaining proof search: we (1) start with the desired endsequent; (2) branch, applying each rule that can be applied; and (3) repeat. This algorithm is trivially complete, because we try all rules. It is also trivially guaranteed to terminate, since a derivation can only use sub-formulas of the formulas in the conclusion—at each successive step, the number of available formulas becomes strictly smaller, and so we will eventually run out of formulas.

The sequent calculus formulation is equivalent to the natural deduction formulation from figure 4. This is trivial to prove once you have a procedure for cut-elimination (see Moot and Retoré, 2012, p. 107). Therefore, we are still able to translate to $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$, and obtain an interpretation. However, in the next section we will discuss the alternative to this sequent calculus that we will use, so we will forgo this exercise.

[4] We use bold-face to distinguish between the variables associated with each word, and the meaning postulates we use in our semantics.

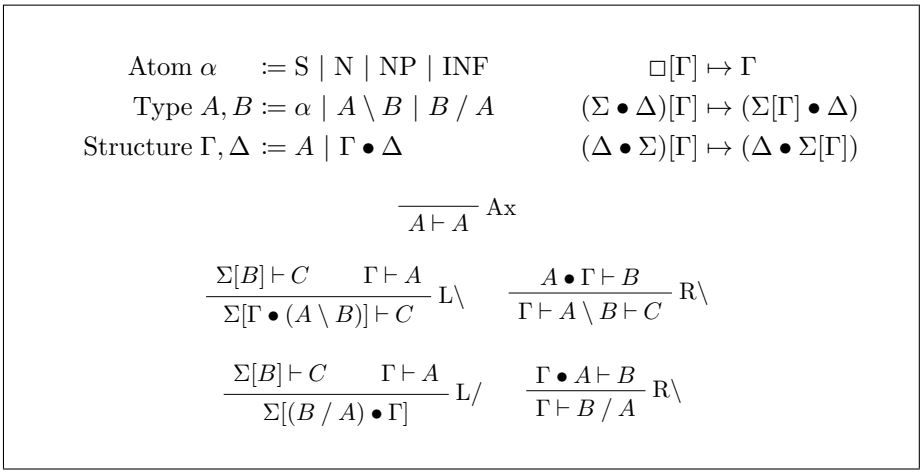


Figure 6: NL (Lambek, 1961) in sequent calculus style.

2 Display calculus and focused proof search

In this section, we will develop a display calculus (Jr., 1982) for NL. We will start out by presenting a display calculus for NL based on work by Moortgat (2009), Bernardi and Moortgat (2010), Moortgat and Moot (2011) and Goré (1998). We will then continue by motivating our choice for display calculus over sequent calculus. In section 2.3 we will relate our display calculus back to the framework discussed in section 1, by defining a translation from our display calculus back to $\lambda^{\rightarrow\times}_{\{\mathbf{e},\mathbf{t}\}}$. And finally, in section 2.4, discuss the problem of spurious ambiguity, and address this by developing an extension to display calculus, using polarities and focusing (Girard, 1991; Bastenhof, 2011), which is free of spurious ambiguity.

2.1 NL as a display calculus

We present the display calculus for NL in figure 7.[5] It features the same atoms and types as in figure 4, but structures have been expanded: there are now positive and negative structures—with one structural connective for each logical connective—and residuation rules to navigate them.[6] These two work together to guarantee the *display property*—the property that any sub-structure can be made the sole structure in either the antecedent or the succedent, depending on its polarity. For instance, below we use residuation to isolate the object NP on the left-hand side:[7]

[5] The calculus in figure 7 was first formulated by Moortgat and Oehrle (1999)—though without display calculus in mind. Because the logical and structural connectives in NL coincide, they did not think to distinguish between them. Therefore, they do not have the R\ and R/ rules, and their version has slightly different properties.

[6] For each connective, we use the *same* symbol at both the logical and the structural level. This is unconventional. However, because each type, when used in a structure, is wrapped in the $\cdot\_\cdot$-connective, this is perfectly unambiguous.

[7] Inverted applications of the residuation rules are marked by switching the operators—e.g. by writing Res$\bullet\backslash$ instead of Res$\backslash\bullet$.

$$
\begin{array}{rl}
\text{Atom} & \alpha \coloneqq \mathrm{S} \mid \mathrm{N} \mid \mathrm{NP} \mid \mathrm{PP} \mid \mathrm{INF} \\
\text{Type} & A, B \coloneqq \alpha \mid A \setminus B \mid B \,/\, A \\
\text{Structure}^{+} & \Gamma \coloneqq \cdot A \cdot \mid \Gamma_1 \bullet \Gamma_2 \\
\text{Structure}^{-} & \Delta \coloneqq \cdot A \cdot \mid \Gamma \setminus \Delta \mid \Delta \,/\, \Gamma
\end{array}
$$

$$
\frac{}{\cdot\alpha\cdot \vdash \cdot\alpha\cdot}\,\mathrm{Ax}
$$

$$
\frac{\Gamma \vdash \cdot A\cdot \qquad \cdot B\cdot \vdash \Delta}{\cdot A \setminus B\cdot \vdash \Gamma \setminus \Delta}\,\mathrm{L}\setminus
\qquad
\frac{\Gamma \vdash \cdot A\cdot \setminus \cdot B\cdot}{\Gamma \vdash \cdot A \setminus B\cdot}\,\mathrm{R}\setminus
$$

$$
\frac{\Gamma \vdash \cdot A\cdot \qquad \cdot B\cdot \vdash \Delta}{\cdot B \,/\, A\cdot \vdash \Delta \,/\, \Gamma}\,\mathrm{L}/
\qquad
\frac{\Gamma \vdash \cdot B\cdot \,/\, \cdot A\cdot}{\Gamma \vdash \cdot B \,/\, A\cdot}\,\mathrm{R}/
$$

$$
\frac{\Gamma_2 \vdash \Gamma_1 \setminus \Delta}{\Gamma_1 \bullet \Gamma_2 \vdash \Delta}\,\mathrm{Res}\setminus\bullet
\qquad
\frac{\Gamma_1 \vdash \Delta \,/\, \Gamma_2}{\Gamma_1 \bullet \Gamma_2 \vdash \Delta}\,\mathrm{Res}/\bullet
$$

Figure 7: NL (Lambek, 1961) as a display calculus.

$$
\dfrac{\dfrac{\vdots}{\cdot\mathrm{NP}\cdot \bullet (\cdot\mathrm{TV}\cdot \bullet \cdot\underline{\mathrm{NP}}\cdot) \vdash \cdot\mathrm{S}\cdot}\,\mathrm{Res}\bullet\setminus}{\cdot\mathrm{TV}\cdot \bullet \cdot\underline{\mathrm{NP}}\cdot \vdash \cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot}\,\mathrm{Res}\bullet/
$$

Because the display property guarantees that any sub-structure can be displayed on either the right- or the left-hand side of the turnstile, we will occasionally abbreviate inferences such as the one above using the display postulate (DP):[8]

$$
\dfrac{\dfrac{\vdots}{\cdot\mathrm{NP}\cdot \bullet (\cdot\mathrm{TV}\cdot \bullet \cdot\underline{\mathrm{NP}}\cdot) \vdash \cdot\mathrm{S}\cdot}\,\mathrm{DP}}{\cdot\underline{\mathrm{NP}}\cdot \vdash (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot) \,/\, \cdot\mathrm{TV}\cdot}
$$

Using the display property, we can trivially derive the sequent calculus rules. For instance, below we derive the sequent-calculus version of L\:[9]

$$
\dfrac{\dfrac{\dfrac{\dfrac{\Sigma[\cdot B\cdot] \vdash \cdot C\cdot}{\cdot B\cdot \vdash \Sigma'[\cdot C\cdot]}\,\mathrm{DP} \qquad \Gamma \vdash \cdot A\cdot}{\cdot A \setminus B\cdot \vdash \Gamma \setminus \Sigma'[\cdot C\cdot]}\,\mathrm{Res}\setminus\bullet}{\Gamma \bullet \cdot A \setminus B\cdot \vdash \Sigma'[\cdot C\cdot]}}{\Sigma[\Gamma \bullet \cdot A \setminus B\cdot] \vdash \cdot C\cdot}\,\mathrm{DP}
$$

[8] We have attached a formal, executable proof of the display property for full NLQ—an extension of display NL as presented in this section. See the appendix A for more details.

[9] Σ' is the representation of Σ after it has been moved by the display postulate. A formal definition of this relation is given in appendix A.

Again, we use the product-free version of NL. Should the reader need a product, though, it is easily added (see Moortgat and Moot, 2011):

$$\frac{\cdot A\cdot \bullet \cdot B\cdot \vdash X}{\cdot A \otimes B\cdot \vdash X}\,\mathrm{L}\otimes \qquad \frac{X \vdash \cdot A\cdot \qquad Y \vdash \cdot B\cdot}{X \bullet Y \vdash \cdot A \otimes B\cdot}\,\mathrm{R}\otimes$$

One change that we made, aside from moving to display calculus, is that the axiom has been restricted to atoms. This does not mean our logic no longer has the identity, since it is derivable by simple induction over the structure of the formula:

$$\frac{}{\cdot\alpha\cdot \vdash \cdot\alpha\cdot}\,\mathrm{Ax} \qquad \frac{\dfrac{\dfrac{\vdots}{\cdot A\cdot \vdash \cdot A\cdot} \quad \dfrac{\vdots}{\cdot B\cdot \vdash \cdot B\cdot}}{\cdot A \setminus B\cdot \vdash \cdot A\cdot \setminus \cdot B\cdot}\,\mathrm{L}\setminus}{\cdot A \setminus B\cdot \vdash \cdot A \setminus B\cdot}\,\mathrm{R}\setminus \qquad \frac{\dfrac{\dfrac{\vdots}{\cdot A\cdot \vdash \cdot A\cdot} \quad \dfrac{\vdots}{\cdot B\cdot \vdash \cdot B\cdot}}{\cdot B \,/\, A\cdot \vdash \cdot B\cdot \,/\, \cdot A\cdot}\,\mathrm{L}\setminus}{\cdot B \,/\, A\cdot \vdash \cdot B \,/\, A\cdot}\,\mathrm{R}/$$

Instead, the change was made to avoid spurious ambiguity. If the calculus *were* to have a full identity, then there would be e.g. *two* proofs of the identity over IV:

$$\frac{}{\cdot\mathrm{NP} \setminus \mathrm{S}\cdot \vdash \cdot\mathrm{NP} \setminus \mathrm{S}\cdot}\,\mathrm{Ax} \qquad \frac{\dfrac{\dfrac{}{\cdot\mathrm{NP}\cdot \vdash \cdot\mathrm{NP}\cdot}\,\mathrm{Ax} \quad \dfrac{}{\cdot\mathrm{S}\cdot \vdash \cdot\mathrm{S}\cdot}\,\mathrm{Ax}}{\cdot\mathrm{NP} \setminus \mathrm{S}\cdot \vdash \cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot}\,\mathrm{L}\setminus}{\cdot\mathrm{NP} \setminus \mathrm{S}\cdot \vdash \cdot\mathrm{NP} \setminus \mathrm{S}\cdot}\,\mathrm{R}\setminus$$

This is problematic. Generally, we only want to have two proofs for the same sequent when that sequent is associated with an ambiguous sentence. The derivation for e.g. "Mary left" contains the above derivation—as NP \ S is the type of 'left'—but this sentence is not ambiguous at all! In fact, when we use the derived identity described above, the two proofs will expand to be equal. The problem of spurious ambiguity is further discussed in section 2.4.

In order for our calculus to be a valid display calculus, it needs to obey eight simple conditions. Of these conditions, the only one that involves any proof burden is **C8**—adapted from Goré (1998):

> If there are inference rules ρ_1 and ρ_2 with respective conclusions $\Gamma \vdash \cdot A\cdot$ and $\cdot A\cdot \vdash \Delta$ and if Cut is applied to yield $\Gamma \vdash \Delta$ then, either $\Gamma \vdash \Delta$ is identical to $\Gamma \vdash \cdot A\cdot$ or to $\cdot A\cdot \vdash \Delta$; or it is possible to pass from the premises of ρ_1 and ρ_2 to $\Gamma \vdash \Delta$ by means of inferences falling under Cut where the cut-formula is always a proper sub-formula of A.

In other words, we have to show that we can rewrite cuts on matching left and right rules to smaller cuts on proper sub-formulas of the cut-formula. For L\ and R\, this is done as follows:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Pi \vdash \cdot A\cdot \setminus \cdot B\cdot}\,\mathrm{R}\setminus}{\Pi \vdash \cdot A \setminus B\cdot} \quad \dfrac{\dfrac{\vdots}{\Gamma \vdash \cdot A\cdot} \quad \dfrac{\vdots}{\cdot B\cdot \vdash \Delta}}{\cdot A \setminus B\cdot \vdash \Gamma \setminus \Delta}\,\mathrm{L}\setminus}{\Pi \vdash \Gamma \setminus \Delta}\,\mathrm{Cut}$$

$$\Longrightarrow$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \cdot A \cdot} \qquad \dfrac{\dfrac{\dfrac{\vdots}{\Pi \vdash \cdot A \cdot \setminus \cdot B \cdot}}{\cdot A \cdot \bullet \Pi \vdash \cdot B \cdot}\,\mathrm{Res}\setminus\bullet \qquad \vdots}{\cdot B \cdot \vdash \Delta}\,\mathrm{Cut}}{\dfrac{\cdot A \cdot \bullet \Pi \vdash \Delta}{\cdot A \cdot \vdash \Delta \,/\, \Pi}\,\mathrm{Res}\bullet/}\,\mathrm{Cut}}{\dfrac{\dfrac{\Gamma \vdash \Delta \,/\, \Pi}{\Gamma \bullet \Pi \vdash \Delta}\,\mathrm{Res}/\bullet}{\Pi \vdash \Gamma \setminus \Delta}\,\mathrm{Res}\bullet\setminus}$$

And likewise for L/ and R/.

2.2 Why use display calculus?

There are a few key advantages to using display calculus. First of all, display calculus generalises sequent calculus. What this means is that if something is a display calculus, it has all the properties commonly associated with sequent calculus. Amongst others, display calculus has the property that we are looking for: it has an easy to implement, complete algorithm for proof search.

However, display calculus is more than sequent calculus. One of the main theorems regarding sequent calculus—Gentzen's 'Hauptsatz'—is the proof of cut-elimination. Whereas for sequent calculus, this theorem has to be proved separately for each instance, display calculus has a generic proof of cut-elimination, which holds whenever the calculus obeys certain easy to check conditions.

One last reason is that display calculus is, due to the way in which it is usually formulated, relatively easy to formalise.

Below we will discuss these arguments in favour of display calculus in more detail.

Practical proof search procedure All display calculi (by definition) have the sub-formula property. However, in our display calculus, we have added structural rules, which do not mention formulas. Therefore, we can no longer guarantee termination—and thus decidability—based on the sub-formula property alone. Display calculi, however, do not necessarily have the *sub-structure* property—the property that a structural rule, in its premises, can only use proper sub-structures of the structures in its conclusion. This makes sense, since many logics depend crucially on inference rules that do not have this property—e.g. contraction and weakening. However, this does mean we will have to take special care that the structural rules we introduce will not break the guarantee of termination.

There is one simple extension we can make to the backward-chaining proof search procedure, which gives us more freedom in the formulation of structural rules. For this, let us have a look at the residuation rules in figure 7. They are crucial to the display calculus, but clearly do not have the sub-structure property: since they can be applied either way around, it is easy to see how they can cause problems with termination. This non-termination, however, is benign; these rules can only cause *loops*—any non-termination caused by them is guaranteed to return to the same sequent. We can extend our proof search procedure with loop-checking to cope with this. We do this by (1) passing along a set of visited sequents; (2) stopping the proof search if we ever visit the same

sequent twice; and (3) emptying out this set if we make progress—where progress means eliminating a connective. Such an extension also preserves completeness, since any proof that has a loop in it can be trivially rewritten to a proof without a loop by cutting out the loop.

The problem then remains to avoid structural rules—or combinations thereof—which can cause a divergence in which *no* sequent is visited more than once. We will discuss this further in section 4.3.1

Generic proof of cut-elimination Another important property of display calculus is the generic proof of cut-elimination. A proof of cut-elimination means that cut is an admissible rule, i.e. that every proof which uses the cut rule can be rewritten to a proof that does not use the cut rule:

$$\frac{\Gamma \vdash A \qquad A \vdash \Delta}{\Gamma \vdash \Delta}\ \text{Cut}$$

This is important, amongst other reasons, because a logic has to admit the cut rule by definition. However, if we were to include cut as an explicit rule, we would no longer be able to use backward-chaining proof search; the cut rule can always be applied, and introduces a unknown formula A.

Another reason why cut is important is because it embodies a linguistic intuition that many of us have: the idea that if you have a sentence which contains a noun phrase—e.g. 'a book' in "Mary met John"—and we have some other phrase of which we known that it is also a noun phrase—e.g. "the tallest man"—then we should be able to substitute that second noun phrase for the first, and the result should still be a grammatical sentence—e.g. "Mary met the tallest man."

It should be clear that it is always important for the cut rule to be admissible. However, in practice, one often has to give a separate proof of cut-elimination for every logic. The generic proof of cut-elimination for display calculus, however, states that if a calculus obeys certain conditions (see Goré, 1998), the cut rule is admissible. This makes it an invaluable tool for research. In this thesis, we will discuss several extensions to the non-associative Lambek calculus. Because we know that each of these extensions respects the rules of display calculus, we can be sure that any combination of then will have a proof of cut-elimination, without having to prove this even once.

Easy to formalise One last property of display calculus that is useful in formalising the calculus, is the fact that display calculus does not rely on the mechanisms of contexts and plugging functions, as used in figure 3 and the usual sequent calculus formulation of NL. These mechanisms are sometimes touted for simplifying the presentation of proofs on paper, and for decreasing the complexity of proof search—the idea being that there are fewer rules to apply.

However, they greatly complicate formal meta-logical proofs using, for instance, proof assistants such as Coq or Agda. For some intuition as to why, note that using contexts generally inserts an application of the plugging function '$\cdot[\cdot]$' in the *conclusions* of inference rules. This means that, in order to do, for instance, a proof by induction on the structure of the sequent, one has a much harder time proving which rules can lead to this sequent. In dependently-typed

programming, the equivalent is inserting function applications in the return types of the constructors of datatypes. In their implementation of verified binary search trees, McBride (2014) notes that this is bad design, as it leads to an increased proof burden.

Another issue is that, on paper, researchers are used to reasoning with a number of simplifications: the function '$\cdot[\cdot]$' is often overloaded to mean "plug structure into context" and "compose two contexts", and we reason up to e.g. the equivalence between the two $(\Sigma_1[\Sigma_2[\Gamma]] \equiv (\Sigma_1[\Sigma_2])[\Gamma])$ and the associativity of context composition. When using a proof assistant, these implicit rewrite steps become painfully obvious.

To make matters worse, it is not trivial to see if these mechanisms actually *do* decrease the size of the proofs. Undoubtedly, there are fewer rule applications, but the flipside of this is that each rule application involving a context must now implicitly be decorated with that context to be perfectly unambiguous. Note that to apply a *single* rule under a context of depth d, we need to write $2d$ applications of residuation rules—$1d$ to move down into the context, $1d$ to move back up—as opposed of annotating with a single context of size d.[10] A doubling in size. However, if we have n successive applications under the *same* context, then residuation starts to take the upper hand. For residuation, we still need to write $2d$ contexts, but for annotated rules, we will need to repeat the context each time, writing contexts of total size nd.

In a similar vein, it is hard to see whether these mechanisms reduce amount of work to be done during proof search. While there are indeed fewer rules, each of these rules can now be applied under a variety of contexts. This last point hints at another advantage of not using contexts: it allows for the proof search algorithm to be truly trivial, as we can say a rule applies if its conclusion can be unified with the current proof obligation, and do not have to check all possible contexts under which this unification could succeed.

2.3 Terms for display NL

In the previous sections, we defined a display calculus which is equivalent to our natural deduction formulation of NL from section 1. However, there is still one thing missing from our new implementation: terms.

We could translate display NL to natural deduction NL, and use the term labelling that is the result of that translation. However, in later sections we will extend display NL to be more expressive, and we do not want to be forced to update the natural deduction formulation as well. In addition, the extra indirection would complicate matters too much. We therefore choose to give a direct translation from display NL to lambda calculus.

There is one problem in translating display NL to the lambda calculus: the structures for display NL are much more expressive. There are structural connectives for implication, and since each logical connective must have a structural equivalent, we will certainly add more structural connectives in later sections. For this reason, we choose to translate all structures to types. The one downside to this is that we must translate the product '$\bullet$' to the product type '$\times$', and

[10] Though contexts are usually conceptualised as "structures with a single hole", there is no need to write anything but the path to the hole—that is to say, it is perfectly unambiguous to write "$\_ \bullet (\square \bullet \_)$" instead of e.g. $\mathrm{NP} \bullet (\square \bullet ((\mathrm{N} \,/\, \mathrm{NP}) \bullet \mathrm{NP}))$. Therefore, we can say *size* d instead of *depth* d.

insert the necessary machinery to pack and unpack these products. In figure 8, we extend our semantic calculus with products. Anticipating future needs, we also extend it with units.

The term labelling for display NL is presented in figure 9. The lambda terms are typed by the translations of the formulas from display NL. As is usual when translating sequent calculus to natural deduction, our term labelling employs substitution, which was defined in section 1.2.

Now that we once again have a complete type-logical grammar, let us take a quick look at an example, "Mary likes Bill":

$$\cdot\text{mary} : \text{NP}\cdot \bullet (\cdot\text{likes} : \text{TV}\cdot \bullet \cdot\text{bill} : \text{NP}\cdot) \vdash ? : \cdot\text{S}\cdot$$

If we search for proofs, using backward-chaining proof search, we find the following proof:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot\text{NP}\cdot \vdash \cdot\text{NP}\cdot}\,\text{Ax} \quad \dfrac{}{\cdot\text{S}\cdot \vdash \cdot\text{S}\cdot}\,\text{Ax}}{\cdot\text{NP} \setminus \text{S}\cdot \vdash \cdot\text{NP}\cdot \setminus \cdot\text{S}\cdot}\,\text{L}\setminus \quad \dfrac{}{\cdot\text{NP}\cdot \vdash \cdot\text{NP}\cdot}\,\text{Ax}}{\cdot(\text{NP} \setminus \text{S}) \,/\, \text{NP}\cdot \vdash (\cdot\text{NP}\cdot \setminus \cdot\text{S}\cdot) \,/\, \cdot\text{NP}\cdot}\,\text{L}/}{\cdot(\text{NP} \setminus \text{S}) \,/\, \text{NP}\cdot \bullet \cdot\text{NP}\cdot \vdash \cdot\text{NP}\cdot \setminus \cdot\text{S}\cdot}\,\text{Res}/\bullet}{\cdot\text{NP}\cdot \bullet (\cdot(\text{NP} \setminus \text{S}) \,/\, \text{NP}\cdot \bullet \cdot\text{NP}\cdot) \vdash \cdot\text{S}\cdot}\,\text{Res}\setminus\bullet$$

Applying the translation from figure 9 gives us the following lambda term:

$$s : \mathbf{e} \times (\mathbf{eet} \times \mathbf{e}) \vdash (\text{case } s \text{ of } (\text{mary}, (\text{likes}, \text{bill})) \to (\text{likes bill}) \text{ mary}) : \mathbf{t}$$

This lambda term takes the sentence structure apart, and computes the meaning. If this is desirable, it is possible to do some post-processing with the structuralisation lemma:

$$\dfrac{A \ \vdash B}{\text{St}(A) \vdash B}\,\text{St} \quad \textbf{where} \quad \begin{array}{lll} \text{St}(A \times B) & \mapsto \text{St}(A), \text{St}(B) \\ \text{St}(\top) & \mapsto \varnothing \\ \text{St}(A) & \mapsto A \end{array}$$

This would result—after β-normalisation—in the following lambda term:

$$\text{mary} : \text{NP}, \text{likes} : \text{TV}, \text{bill} : \text{NP} \vdash ((\text{likes bill}) \text{ mary}) : \mathbf{t}$$

The lemma itself is fairly easy to derive by induction on the antecedent. Using it has the advantage that the lambda term takes the lexical definitions—the values for mary, likes and bill—from a *linear* structure, instead of from a nested tuple.

2.4 Focusing and spurious ambiguity

In section 2.1, we briefly touched upon spurious ambiguity. We investigated the spurious ambiguity inherent in the axiom, but we never gave a formal definition of spurious ambiguity.

To be able to give such a formal definition, we need the notion of a *normal-form*. For instance, Andreoli (1992) defines a normal-form for full linear logic. This system removes e.g. the choice in when to apply contraction and weakening, and in which order to decompose the formulae. Hepple (1990) does a similar

thing for the Lambek calculus. Once we have a normal-form, we can define spurious ambiguity in our search procedure as finding multiple proofs for which the normal-forms are equal.

We can refine the above definition of spurious ambiguity, by referring to the semantic interpretation (Capelletti, 2007; Bastenhof et al., 2013). Below is a diagram representing the function which interprets syntactic terms, where '$(\cdot)^*$' is the translation from NL to $\lambda^{\rightarrow,\times}_{\{\mathbf{e},\mathbf{t}\}}$, and 'lex' is the function that inserts the lexical definitions for words:[11]

$$\mathrm{NL} \xrightarrow{(\cdot)^*} \lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}} \xrightarrow{\mathrm{lex}} \lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$$

We can define spurious ambiguity with respect to the second node, i.e. the semantic terms after applying the translation '$(\cdot)^*$' *and normalising*, but before inserting the lexical definitions. In effect, we want a normal-form for NL which is guaranteed to translate to a *unique* normal-form term in $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$. Moortgat and Moot (2011) define such a normal-form for LG, for which we present the NL fragment in figure 10.

Our focused display calculus is a system with *three* kinds of sequents: the original structural sequent $\Gamma \vdash \Delta$, and two focused sequents $\boxed{A} \vdash \Delta$ and $\Gamma \vdash \boxed{B}$. There are four new rules which communicate between these different sequents: left and right focusing and unfocusing. In addition, the axiom is split into a left- and a right-focused axiom. The crucial point is that all formulas are assigned a polarity, and that the axioms and the focusing and unfocusing rules are restricted to formulas of certain polarities, forcing the proofs into a normal form. For complex types, this polarity is based on the main connective, but for atomic formulas, polarities are assigned. The choice of polarity affects the kinds of ambiguity that we allow—this will be discussed further in section 4.2.[12]

Let us discuss an example. In the unfocused display calculus, there are two proofs associated with the sentence "Mary saw the fox":

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\cdot \mathrm{N} \cdot \vdash \cdot \mathrm{N} \cdot}\,\mathrm{Ax} \quad \dfrac{}{\cdot \mathrm{NP} \cdot \vdash \cdot \mathrm{NP} \cdot}\,\mathrm{Ax}}{\cdot \mathrm{NP} \,/\, \mathrm{N} \cdot \vdash \cdot \mathrm{NP} \cdot \,/\, \cdot \mathrm{N} \cdot}\,\mathrm{L}/}{\cdot \mathrm{NP} \,/\, \mathrm{N} \cdot \bullet \cdot \mathrm{N} \cdot \vdash \cdot \mathrm{NP} \cdot}\,\mathrm{Res}/\bullet \quad \dfrac{\dfrac{}{\cdot \mathrm{NP} \cdot \vdash \cdot \mathrm{NP} \cdot}\,\mathrm{Ax} \quad \dfrac{}{\cdot \mathrm{S} \cdot \vdash \cdot \mathrm{S} \cdot}\,\mathrm{Ax}}{\cdot \mathrm{NP} \setminus \mathrm{S} \cdot \vdash \cdot \mathrm{NP} \cdot \,/\, \cdot \mathrm{S} \cdot}\,\mathrm{L}\setminus}{\cdot (\mathrm{NP} \setminus \mathrm{S}) \,/\, \mathrm{NP} \cdot \vdash (\cdot \mathrm{NP} \cdot \setminus \cdot \mathrm{S} \cdot) \,/\, (\cdot \mathrm{NP} \,/\, \mathrm{N} \cdot \bullet \cdot \mathrm{N} \cdot)}\,\mathrm{L}/}{\cdot (\mathrm{NP} \setminus \mathrm{S}) \,/\, \mathrm{NP} \cdot \bullet \cdot \mathrm{NP} \,/\, \mathrm{N} \cdot \bullet \cdot \mathrm{N} \cdot \vdash \cdot \mathrm{NP} \cdot \setminus \cdot \mathrm{S} \cdot}\,\mathrm{Res}/\bullet}{\cdot \mathrm{NP} \cdot \bullet \cdot (\mathrm{NP} \setminus \mathrm{S}) \,/\, \mathrm{NP} \cdot \bullet \cdot \mathrm{NP} \,/\, \mathrm{N} \cdot \bullet \cdot \mathrm{N} \cdot \vdash \cdot \mathrm{S} \cdot}\,\mathrm{Res}\setminus\bullet$$

[11] Note that the second node is often considered to be *linear* $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$, i.e. the fragment of $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$ without contraction and weakening. This is because the syntactic calculus is not supposed to do copying or deletion, and we may want to limit non-linearity to lexical semantics.

[12] For the remainder of this thesis, we will discuss extension in their focused form, and include the polarities of any new type. Should you wish to remove focusing, you can simply rewrite the focused sequents to structural sequents (i.e. $\boxed{A} \equiv \cdot A \cdot$), merge identifal rules, and remove any rules that become the identity.

$$\dfrac{\dfrac{}{\cdot\mathrm{N}\cdot \vdash \cdot\mathrm{N}\cdot}\,\mathrm{Ax} \qquad \dfrac{\dfrac{\dfrac{}{\cdot\mathrm{NP}\cdot \vdash \cdot\mathrm{NP}\cdot}\,\mathrm{Ax} \qquad \dfrac{\dfrac{}{\cdot\mathrm{NP}\cdot \vdash \cdot\mathrm{NP}\cdot}\,\mathrm{Ax} \qquad \dfrac{}{\cdot\mathrm{S}\cdot \vdash \cdot\mathrm{S}\cdot}\,\mathrm{Ax}}{\cdot\mathrm{NP} \setminus \mathrm{S}\cdot \vdash \cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot}\,\mathrm{L}\setminus}{\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \vdash (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot) / \cdot\mathrm{NP}\cdot}\,\mathrm{L}/}{\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \bullet \cdot\mathrm{NP}\cdot \vdash (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}\,\mathrm{Res}\bullet/}{\cdot\mathrm{NP}\cdot \vdash \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}\,\mathrm{Res}\setminus\bullet}{\cdot\mathrm{NP} / \mathrm{N}\cdot \vdash (\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)) / \cdot\mathrm{N}\cdot}\,\mathrm{L}/$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot\mathrm{NP} / \mathrm{N}\cdot \bullet \cdot\mathrm{N}\cdot \vdash \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}{\cdot\mathrm{NP} / \mathrm{N}\cdot \vdash (\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)) / \cdot\mathrm{N}\cdot}\,\mathrm{Res}/\bullet}{\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \bullet \cdot\mathrm{NP} / \mathrm{N}\cdot \bullet \cdot\mathrm{N}\cdot \vdash \cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot}\,\mathrm{Res}\setminus\bullet}{\cdot\mathrm{NP}\cdot \bullet \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \bullet \cdot\mathrm{NP} / \mathrm{N}\cdot \bullet \cdot\mathrm{N}\cdot \vdash \cdot\mathrm{S}\cdot}\,\mathrm{Res}\setminus\bullet$$

These proofs are clearly distinct, but they are associated with the *same* function-argument structure, (saw (the fox) mary). This is problematic, since we only want ambiguity in our syntax when there is ambiguity in the meaning.

However, if attempt to write these two proofs down in the focused display calculus from figure 10, we see that the first proof can easily be transcribed—inserting focusing and unfocusing rules where needed—but the second proof is rejected, as it contains a polarity error:

$$\dfrac{\dfrac{}{\boxed{\mathrm{N}} \vdash \cdot\mathrm{N}\cdot}\,\mathrm{Ax}^L}{\dfrac{\cdot\mathrm{N}\cdot \vdash \cdot\mathrm{N}\cdot}{\cdot\mathrm{N}\cdot \vdash \boxed{\mathrm{N}}}\,\mathrm{Unf}^R}\,\mathrm{Foc}^L \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\boxed{\mathrm{NP}} \vdash \cdot\mathrm{NP}\cdot}\,\mathrm{Ax}^L}{\cdot\mathrm{NP}\cdot \vdash \cdot\mathrm{NP}\cdot}\,\mathrm{Foc}^L \qquad \dfrac{}{\boxed{\mathrm{S}} \vdash \cdot\mathrm{S}\cdot}\,\mathrm{Ax}^L}{\cdot\mathrm{NP}\cdot \vdash \boxed{\mathrm{NP}}}\,\mathrm{Unf}^R}{\boxed{\mathrm{NP} \setminus \mathrm{S}} \vdash \cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot}\,\mathrm{L}\setminus}{\boxed{(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}} \vdash (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot) / \cdot\mathrm{NP}\cdot}\,\mathrm{L}/}{\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \vdash (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot) / \cdot\mathrm{NP}\cdot}\,\mathrm{Foc}^L$$

$$\dfrac{\dfrac{\dfrac{\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \bullet \cdot\mathrm{NP}\cdot \vdash (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}{\cdot\mathrm{NP}\cdot \vdash \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}\,\mathrm{Res}\setminus\bullet}{\boxed{\mathrm{NP}} \vdash \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}\,\mathrm{Unf}^L \leftarrow \text{wrong}}{\boxed{\mathrm{NP} / \mathrm{N}} \vdash (\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)) / \cdot\mathrm{N}\cdot}\,\mathrm{L}/$$

$$\dfrac{\dfrac{\dfrac{\dfrac{\cdot\mathrm{NP} / \mathrm{N}\cdot \bullet \cdot\mathrm{N}\cdot \vdash \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)}{\cdot\mathrm{NP} / \mathrm{N}\cdot \vdash (\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \setminus (\cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot)) / \cdot\mathrm{N}\cdot}\,\mathrm{Res}/\bullet}{\cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \bullet \cdot\mathrm{NP} / \mathrm{N}\cdot \bullet \cdot\mathrm{N}\cdot \vdash \cdot\mathrm{NP}\cdot \setminus \cdot\mathrm{S}\cdot}\,\mathrm{Res}\setminus\bullet}{\cdot\mathrm{NP}\cdot \bullet \cdot(\mathrm{NP} \setminus \mathrm{S}) / \mathrm{NP}\cdot \bullet \cdot\mathrm{NP} / \mathrm{N}\cdot \bullet \cdot\mathrm{N}\cdot \vdash \cdot\mathrm{S}\cdot}\,\mathrm{Res}\setminus\bullet$$

Had we chosen differently, and assigned NP a positive polarity, this proof would be valid. However, in that case the *first* proof would have contained a polarity error. We leave it as an exercise for the reader to find this error.

Looking at the proof above—especially at the three left-most axioms—one might wonder why we bother restricting the polarity for axioms at all, since obviously the focusing and unfocusing rules allow us to derive left- or right-focused sequents for any atom, regardless of polarity. Once again, this is to avoid spurious ambiguity: if we did not restrict the axioms, there would be two ways to derive either axiom. The above proof has 4 axioms, so—if it were correct—that would result in 2^4 proofs.

Extending display NL with focusing raises one problem: does this extension still obey the conditions for display calculus? We assume that cut has the following form:

$$\dfrac{\Gamma \vdash \boxed{A} \qquad \boxed{A} \vdash \Delta}{\Gamma \vdash \Delta}\,\text{Cut}$$

However, when we try to extend the proof for **C8**, we find that—keeping in mind that we may want to add new connectives or change the polarities of the atomic formulas—there are two places in the proof where we cannot guarantee the polarity of certain formulas:

$$\dfrac{\dfrac{\dfrac{\vdots}{\Pi \vdash \cdot A \cdot \setminus \cdot B \cdot}}{\dfrac{\Pi \vdash \cdot A \setminus B \cdot}{\Pi \vdash \boxed{A \setminus B}}\,\text{Unf}^R} \qquad \dfrac{\vdots \qquad \vdots}{\dfrac{\Gamma \vdash \boxed{A} \qquad \boxed{B} \vdash \Delta}{\boxed{A \setminus B} \vdash \Gamma \setminus \Delta}\,\text{L}\setminus}}{\Pi \vdash \Gamma \setminus \Delta}\,\text{Cut}$$

$$\Longrightarrow$$

$$\dfrac{\dfrac{\vdots}{\Gamma \vdash \boxed{A}} \qquad \dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Pi \vdash \cdot A \cdot \setminus \cdot B \cdot}}{\cdot A \cdot \bullet \Pi \vdash \cdot B \cdot}\,\text{Res}\setminus\bullet}{\cdot A \cdot \bullet \Pi \vdash \boxed{B}}\,\text{Unf}^R \qquad \vdots}{\boxed{B} \vdash \Delta}\,\text{Cut}}{\cdot A \cdot \bullet \Pi \vdash \Delta}\,\text{Res}\bullet/}{\cdot A \cdot \vdash \Delta \,/\, \Pi}\,\text{Unf}^L \leftarrow \text{wrong?}}{\boxed{A} \vdash \Delta \,/\, \Pi}\,\text{Cut}$$

$$\text{wrong?} \rightarrow$$

$$\dfrac{\dfrac{\Gamma \vdash \Delta \,/\, \Pi}{\Gamma \bullet \Pi \vdash \Delta}\,\text{Res}/\bullet}{\Pi \vdash \Gamma \setminus \Delta}\,\text{Res}\bullet\setminus$$

If we assume that we can cut either on an $\boxed{A}$, or on an $\cdot A \cdot$, then we can construct proofs of **C8**; at the places where we cannot guarantee the polarity of A or B, we then have a choice. For instance, in the position where we cut on B, if B is negative, we use the proof as written, but if B is positive, we use Unf^L on the *other* argument of cut, and cut on $\cdot B \cdot$. It is, however, uncertain if this will works within the framework of display calculus, as it was formulated with a single turnstile. This means we can no longer be certain that focused NL is a display calculus, which in turn means that we lose our generic proof of cut-elimination.

For CNL, Bastenhof (2011) solves this problem by proving that there is a normalisation procedure from display CNL to focused CNL. Together with the trivial injection from focused CNL into display CNL, we can show that the two are equivalent. Thus, focused CNL is a display calculus by virtue of being equivalent to display CNL. However, in order to define this procedure, Bastenhof (2011) requires a cut-elimination procedure. As one of our main motives for using display calculus is the generic proof of the admissibility of cut, one can see the futility of this exercise.

Nonetheless, the work by Bastenhof does assure us that focused NL is indeed a display calculus, and indeed enjoys an admissible cut. If we make extensions

to display NL beyond CNL, there is another body of work we can lean on: Andreoli (1992) gives a cut-elimination procedure for focused *full linear logic*. Since any extension that we propose in this thesis is less expressive than linear logic, we believe that between these two results, anything we propose in this thesis should have a valid focusing regime.[13]

Because it does not fall within the scope of this thesis, we will leave the problem of finding an elegant solution to integrating focusing and display calculus as future work. For the remainder of this thesis, we will *assume* that focused NLQ is complete with respect to display NLQ. However, we will continue to give proofs of Jr.'s (1982) **C8** using display NLQ, so that—should we turn out to have made an error our the definition of our focusing regime—the reader can simply remove the focusing regime, and be left with a valid display calculus.

3 Lexical Ambiguity

In this section, we will introduce '&' (additive conjunction), and show how this can be used to encode lexical ambiguity.

The original framework for categorial grammar (Lambek, 1958) already had machinery in place to deal with ambiguity; it allowed for multiple lexical entries for each word. However, in the spirit of wanting to deal with linguistic phenomena in a logical manner, it seems to make more sense to absorb lexical ambiguity into the logic itself. Lambek (1961, p. 170) already does this—he adds the additive conjunction, which he writes '∩'. In figure 11, we describe the same connective, but define it within the framework of focused display calculus. The notation '&' comes from Girard (1987).

Again, we have to prove condition **C8**, in order to show that this extension is compatible with display calculus. This time, the proof is even easier. For L&$_1$ and R&:

$$\dfrac{\dfrac{\begin{array}{cc}\vdots & \vdots\\ \Gamma \vdash \cdot A \cdot & \Gamma \vdash \cdot B \cdot\end{array}}{\Gamma \vdash \cdot A \,\&\, B \cdot}\,\mathrm{R\&} \qquad \dfrac{\vdots}{\cdot A \cdot \vdash \Delta}}{\Gamma \vdash \Delta}\,\mathrm{Cut}\ \dfrac{}{\cdot A \,\&\, B \cdot \vdash \Delta}\,\mathrm{L\&_1}$$

$$\Longrightarrow$$

$$\dfrac{\begin{array}{cc}\vdots & \vdots\\ \Gamma \vdash \cdot A \cdot & \cdot A \cdot \vdash \Delta\end{array}}{\Gamma \vdash \Delta}\,\mathrm{Cut}$$

And likewise for L&$_2$ and R&.

When is this extension useful? Imagine a word like 'want'. This can be used in two different ways, with two different meanings:

"Mary wants John to leave."	"Mary wants to leave."
$\mathbf{want}(\mathbf{mary}, \mathbf{leave}(\mathbf{john}))$	$\mathbf{want}(\mathbf{mary}, \mathbf{leave}(\mathbf{mary}))$

'Wants' has an implicit reflexive object: if no object is explicitly given, it is assumed to be reflexive. Other words that show this behaviour are words such

[13] Not accounting for any errors we may have made in formulating these regimes.

as ‘to wash’ and ‘to shave’. Using our new connective, we can give a single definition for such words, which takes this ambiguity into account:

$$\begin{aligned}&\text{wants} \;\; : (((\text{IV} \,/\, \text{INF} \,/\, \text{NP})) \,\&\, (\text{IV} \,/\, \text{INF}))^*\\&\text{wants} = ((\lambda y\ f\ x.\mathbf{want}(x, f\ y)), (\lambda f\ x.\mathbf{want}(x, f\ x)))\end{aligned}$$

For a detailed discussion of the expressiveness of Lambek calculi enriched with ‘&’ (and its dual, ‘$\oplus$’) we refer the reader to Kanazawa (1992).

Note that figure 11 does not present a focusing regime. This is because, in our formulation, the focusing regime for additives is significantly more complicated than the regime for multiplicatives—perhaps an artefact of the fact that it was formulated for a calculus with only multiplicatives. At any rate, below we present a focusing regime for ‘&’, based on recent work by Morrill and Valentín (2015):

$$\text{if Pol}(A) \;=\; + \left\{ \begin{array}{c|c} \dfrac{\cdot A \cdot \vdash \Delta}{\boxed{A \,\&\, B} \vdash \Delta}\,\text{L\&}_1 & \dfrac{\boxed{A} \vdash \Delta}{\boxed{A \,\&\, B} \vdash \Delta}\,\text{L\&}_1 \\[2ex] \dfrac{\cdot B \cdot \vdash \Delta}{\boxed{A \,\&\, B} \vdash \Delta}\,\text{L\&}_2 & \dfrac{\boxed{B} \vdash \Delta}{\boxed{A \,\&\, B} \vdash \Delta}\,\text{L\&}_2 \end{array} \right\} \text{if Pol}(A) \;=\; -$$

$$\frac{\Gamma \vdash \cdot A \cdot \qquad \Gamma \vdash \cdot B \cdot}{\Gamma \vdash \cdot A \,\&\, B \cdot}\,\text{R\&} \qquad \frac{\boxed{A} \vdash \cdot B \cdot \qquad \boxed{A} \vdash \cdot C \cdot}{\boxed{A} \vdash \cdot B \,\&\, C \cdot}\,\text{R\&}$$

The R&-rules expose a restriction in our syntax: because we encode focusing as left- and right-focused sequents, we cannot abstract over it, and therefore need two copies of the R& rule. On the other hand, Morrill and Valentín (2015) do not seem to have a syntactic restriction that ensures that at most one formula is be focused at a time, and the approach taken by Laurent (2004) critically depends on allowing empty structures—something that we do not want to allow in our grammar logic (see section 1).

At the moment it is unclear how to extend Moortgat and Moot’s (2011) CPS-semantics to include additives.[14]

4 Logical approaches to movement

In this section, we will discuss logical approaches to quantification. We will start out by describing the problem of quantification. Then we will discuss the choice between semantic and syntactic approaches. Next, we will discuss the existing semantic approaches to quantifier raising, and why they are inadequate. We will then discuss various syntactic approaches to quantifier raising, and finish by making our own contributions.

[14] It is for this reason that we do not include additives in our Agda formalisation in appendix A, since we want to demonstrate CPS-semantics.

4.1 Quantification in natural language

Quantification is the problem of analysing scope-taking expressions in natural language. For instance, the canonical interpretation for a sentence such as "Everyone laughs" is:

$$\forall x.\mathbf{person}(x) \supset \mathbf{laugh}(x)$$

The problem here, is that the quantifier "everyone" is ostensibly an NP. However, in the given semantics, it is taking scope over "laughs". A more obvious version of this problem can be seen in "John [saw everyone]", where the quantifier is more deeply nested in the parse tree:

$$\forall y.\mathbf{person}(y) \supset \mathbf{past}(\mathbf{see}(\mathbf{john}, y))$$

The problem becomes even more interesting when you consider sentences with multiple quantifiers. Here we observe a phenomenon known as *scope ambiguity*. The canonical example is the phrase "Everyone loves someone", which has the following interpretations:

$$\forall x.\mathbf{person}(x) \supset \exists y.\mathbf{person}(y) \wedge \mathbf{like}(x, y)$$
$$\exists y.\mathbf{person}(y) \wedge \forall x.\mathbf{person}(x) \supset \mathbf{like}(x, y)$$

There are several easy ways to obtain these semantics within the type-logical grammar that we have established in section 1 and section 3. One approach, due to Montague (1973), is to assign NP the semantic type $(\mathbf{et})\mathbf{t}$. This way, we can define the lexicon as follows:

$$\begin{aligned} \text{john} &= \lambda k.k\ \mathbf{john} \\ \text{everyone} &= \lambda k.\forall x.\mathbf{person}(x) \supset k\ x \\ \text{laughs} &= \lambda x.x\ \mathbf{laugh} \end{aligned}$$

We can even build in scope ambiguity by assigning "loves" an ambiguous type—e.g., using the extension from section 3, we assign it the type TV & TV, and define it as:

$$\text{loves} = \lambda k_2.\lambda k_1.(k_1\ (\lambda x.k_2\ (\lambda y.\mathbf{love}(x, y))), k_2\ (\lambda y.k_1\ (\lambda x.\mathbf{love}(x, y))))$$

However, somehow it feels wrong to manually build scope ambiguity into every verb: it is a universal property of language, so it should not be encoded in the lexicon, but in our type-logical grammar.

It is also quite odd that we give "john"—which by all measures is not a quantifier—the freedom to act as a quantifier. And because we do not distinguish between quantificational and non-quantificational NPs, we are left with a large amount of scope ambiguity—spurious ambiguity. For instance, the sentences "Mary likes John" will be considered ambiguous, with both interpretations equal to $\mathbf{like}(\mathbf{mary}, \mathbf{john})$. Hendriks (1993) a solution to this problem, developing a framework in which terms are lifted into quantificational terms only when this is needed.

Another approach—popular in associative Lambek calculi—is to use higher-order syntactic types. For instance, we can assign "everyone" the syntactic type S / IV. This also gives us the semantic type $(\mathbf{et})\mathbf{t}$, and therefore we are able

to assign it the same term as before. This works perfectly for sentences with subject-quantifiers, and it *does* distinguish between quantificational and non-quantificational NPs, but it has one downside: it does not work for sentences with object-quantifiers. In the *associative* Lambek calculus, we can use the similar looking type (S / NP) \ S—a similarity which extended the popularity of the associative Lambek calculus way past its due date. Meanwhile, in NL, we have to devise a type which actually reflects the sentence structure, such as TV \ IV. Using this type, we can assign "everyone" a second interpretation:

$$\text{everyone} = \lambda k.\lambda x.\forall y.\mathbf{person}(y) \supset (k\ y\ x)$$

The need for multiple definitions for 'everyone' aside, there are more problems with this approach: with our current definitions for "everyone"—and similar definitions for "someone"—we can only derive the first of the two expected interpretations for "Everyone likes someone". Now, we could imagine giving a third definition for "everyone" and "someone", with the type (TV\((S/IV)\IV)), which would allow us to define object quantifiers taking scope over subject quantifiers. However, there are many other cases—think of ditransitive verbs, or verb phrases modified by some number of adverbs. Clearly, this is not an elegant solution, as we are basically hard-coding the structures that quantifiers can take scope over in their types.

One interesting aspect of the two approaches we have discussed so far is that they very clearly divide in a semantic and a syntactic approach. The first approach was implemented in the translation to semantic types, and in the lexicon. The second approach was implemented entirely within the syntactic calculus. With any linguistic phenomenon this is an important question: do we consider it to be syntactic or semantic in nature? With quantification, the community seems divided. In this thesis, we will approach the problem of quantification as a syntactic phenomenon. Therefore, in the next section we will discuss some of the approaches to quantification as a semantic problem, and their limitations. Following this, we will discuss various approaches to quantification as a syntactic problem, discuss their weaknesses, and try to amend these.

4.2 Semantic approaches to scope taking

4.2.1 Continuation-passing style translation

Barker (2002, 2004) advocates the use of continuations in natural language semantics. He does this with several case studies, one of which is quantification. The gist of his story is as follows:

As mentioned before, in the sentence "John [saw everyone]", the deeply nested quantifier "everyone" must take scope over the remainder of the sentence. Instead of translating NP to the higher-order type (**et**)**t**, Barker proposes to lift *every type*, consistently, using a technique from computer science known as continuation-passing style (CPS) translation. After translating the syntactic terms, he applies a CPS-translation, lifting expressions of type A into the type $(A \to R) \to R$ for some answer type R. He assumes that the function-argument structure that is the result of the translation to syntactic terms is constructed

using only variables and function application,[15] and defines the translation as follows:[16]

$$\begin{aligned}\overline{x} &= x\\ \overline{M\ N} &= \lambda k.\overline{N}\ (\lambda n.\overline{M}\ (\lambda m.k\ (m\ n)))\end{aligned}$$

This translation is applied to the function-argument structure *before* lexical definitions are inserted. In addition, the lexical items have to be lifted manually for this to work—although the lifting is a much simpler process:

$$\begin{aligned}\text{john} &= \lambda k.k\ \mathbf{john}\\ \text{everyone} &= \lambda k.\forall x.\mathbf{person}(x) \supset k\ x\\ \text{laughs} &= \lambda k.k\ \mathbf{laugh}\\ \text{loves} &= \lambda k.k\ \mathbf{love}\end{aligned}$$

However, this analysis of quantification has several issues. The first of these is already mentioned by Barker (2004): scope ambiguity. The solution provided by Barker (2002, 2004) will only derive the surface-scope interpretation of "Everyone loves someone". Barker solves this problem by making the CPS-translation ambiguous, adding another possible translation for function application:

$$\begin{aligned}\overline{x} &= x\\ \overline{M\ N} &= \lambda k.\overline{N}\ (\lambda n.\overline{M}\ (\lambda m.k\ (m\ n)))\\ \overline{M\ N} &= \lambda k.\overline{M}\ (\lambda m.\overline{N}\ (\lambda n.k\ (m\ n)))\end{aligned}$$

This does solve the problem at hand, and derives both the surface-scope and the inverse-scope interpretations. However, this solution results in a huge amount of spurious ambiguity. A sentence with n words has $n-1$ function applications, and will therefore have $2^{(n-1)}$ interpretations. But this ambiguity is only relevant for scope-takers. In the motivating example, "loves" is not a scope taker. Nevertheless, it is treated as one, which means there are two possible translations for "loves someone", resulting in four interpretations for the sentence where we only want two. This ambiguity grows exponentially with the length of the sentence.

4.2.2 Focused, polarised semantics

Instead of performing a separate CPS translation on the semantic terms, Bernardi and Moortgat (2010), Bastenhof (2010), Moortgat and Moot (2011)—based on work by Girard (1991) and Curien and Herbelin (2000)—integrate the CPS-semantics into the syntactic calculus. The result of this work is the focused Lambek-Grishin (LG) calculus, a bilinear variant of NL, with semantics inspired on Parigot's (1992) $\lambda\mu$-calculus. Focused NL, as presented in this thesis, is a fragment of focused LG. Below we will present the CPS-semantics for the the

[15] This is equivalent to restricting the syntactic calculus to the applicative, i.e. Ajdukiewicz-Bar-Hillel fragment (see Moot and Retoré, 2012, ch. 1.1-1.5, AB-grammars).

[16] It should be noted that Barker's initial solution uses some directional information to ensure that scope-takers are always processed in a left-to-right order, instead of always processing the argument first, as it is presented here. This distinction, however, becomes irrelevant once he introduces the ambiguous translation, and therefore I have chosen not to include it.

product-free NL fragment of focused LG. First of all, the translation on types. Let $A^R = A \to R$:

$$\begin{array}{llll}
[\![\alpha]\!]^+ & = \begin{cases} \alpha^* & \text{if } \mathrm{Pol}(\alpha) \equiv + \\ ((\alpha^*)^R)^R & \text{if } \mathrm{Pol}(\alpha) \equiv - \end{cases} & [\![\alpha]\!]^- & = (\alpha^*)^R \\
[\![A \setminus B]\!]^+ & = ([\![A]\!]^+ \times [\![B]\!]^-)^R & [\![A \setminus B]\!]^- & = [\![A]\!]^+ \times [\![B]\!]^- \\
[\![B \,/\, A]\!]^+ & = ([\![B]\!]^- \times [\![A]\!]^+)^R & [\![B \,/\, A]\!]^- & = [\![B]\!]^- \times [\![A]\!]^+
\end{array}$$

We extend this translation to structures by translating *all* structural connectives as product types. We translate atomic *positive* structures using $[\![\cdot]\!]^+$, and atomic *negative* structures using $[\![\cdot]\!]^-$. We extend it to sequents as follows:

$$\begin{array}{ll}
[\![\ \Gamma \vdash \Delta\ ]\!] & = [\![\Gamma\ ]\!] \vdash [\![\Delta]\!]^R \\
[\![\boxed{A} \vdash \Delta\ ]\!] & = [\![\Delta]\!] \vdash [\![A]\!]^- \\
[\![\ \Gamma \vdash \boxed{B}\ ]\!] & = [\![\Gamma\ ]\!] \vdash [\![B]\!]^+
\end{array}$$

As the translations on sequents and types makes the CPS-semantics for focused NL a somewhat obtuse, we will present them a little more explicitly that usual. Below, we will give the full derivations for the translations of the focusing and unfocusing rules, and some examples for the remaining rules.

Note that for *positive* A, $[\![A]\!]^- \equiv ([\![A]\!]^+)^R$, and for *negative* A, $[\![A]\!]^+ \equiv ([\![A]\!]^-)^R$. Using these facts, we can give the focusing and defocusing rules a valid term labelling. In fact, the left focusing and unfocusing rules are the *only* rules whose translation involves abstraction and application:

Foc^R:

$$\dfrac{\dfrac{\dfrac{[\![x : \Gamma \vdash \boxed{M : A}]\!]}{x : [\![\Gamma]\!] \vdash M : [\![A]\!]^+}}{x : [\![\Gamma]\!] \vdash M : [\![A]\!]^{-R}} \equiv}{[\![x : \Gamma \vdash M : {\cdot}A{\cdot}]\!]}$$

Foc^L:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{k : [\![A]\!]^{-R} \vdash k : [\![A]\!]^{-R}}\,\mathrm{Ax} \quad \dfrac{[\![\boxed{M : A} \vdash x : \Delta]\!]}{[\![x : \Delta]\!] \vdash [\![M : A]\!]^-}}{k : [\![A]\!]^{-R} \bullet x : [\![\Delta]\!] \vdash (k\ M) : R}\,{\to}\mathrm{E}}{k : [\![A]\!]^{-R} \vdash (\lambda x.k\ M) : [\![\Delta]\!]^R}\,{\to}\mathrm{I}}{k : [\![A]\!]^+ \vdash (\lambda x.k\ M) : [\![\Delta]\!]^R} \equiv}{[\![k : {\cdot}A{\cdot} \vdash (\lambda x.k\ M) : \Delta]\!]}$$

Unf^R:

$$\dfrac{\dfrac{\dfrac{[\![x : \Gamma \vdash {\cdot}M : A{\cdot}]\!]}{[\![x : \Gamma]\!] \vdash M : [\![A]\!]^{-R}}}{[\![x : \Gamma]\!] \vdash M : [\![A]\!]^+} \equiv}{[\![x : \Gamma \vdash \boxed{M : A}]\!]}$$

Unf^L:

$$\dfrac{\dfrac{\dfrac{\dfrac{[\![x : {\cdot}A{\cdot} \vdash M : \Delta]\!]}{x : [\![A]\!]^+ \vdash M : [\![\Delta]\!]^R} \quad \dfrac{}{y : [\![\Delta]\!] \vdash y : [\![\Delta]\!]}\,\mathrm{Ax}}{x : [\![A]\!]^+ \bullet y : [\![\Delta]\!] \vdash (M\ y) : R}\,{\to}\mathrm{E}}{y : [\![\Delta]\!] \bullet x : [\![A]\!]^+ \vdash (M\ y) : R}\,\mathrm{Comm.}}{y : [\![\Delta]\!] \vdash [\![(\lambda x.M\ y) : A]\!]^{+R}}\,{\to}\mathrm{I}}{y : [\![\Delta]\!] \vdash (\lambda x.M\ y) : [\![A]\!]^-} \equiv}{[\![\boxed{(\lambda x.M\ y) : A} \vdash y : \Delta]\!]}$$

As for the rest of the rules: both of the axioms, and both the right rules, simply translate to the identity function. The left rules create pairs. For instance, L$\setminus$, translates as follows:

$$\dfrac{\dfrac{\dfrac{}{z : [\![\Gamma]\!] \times [\![\Delta]\!] \vdash z : [\![\Gamma]\!] \times [\![\Delta]\!]}\,\mathrm{Ax} \quad \dfrac{\dfrac{[\![x : \Gamma \vdash \boxed{M : A}]\!]}{x : [\![\Gamma]\!] \vdash M : [\![A]\!]^+} \quad \dfrac{[\![\boxed{N : B} \vdash y : \Delta]\!]}{y : [\![\Delta]\!] \vdash N : [\![B]\!]^-}\,\times\mathrm{I}}{x : [\![\Gamma]\!] \bullet y : [\![\Delta]\!] \vdash (M, N) : [\![A]\!]^+ \times [\![B]\!]^-}}{z : [\![\Gamma]\!] \times [\![\Delta]\!] \vdash (\text{case } z \text{ of } (x, y) \to (M, N)) : [\![A]\!]^+ \times [\![B]\!]^-}\,\times\mathrm{E}}{[\![\boxed{(\text{case } z \text{ of } (x, y) \to (M, N)) : A \setminus B} \vdash z : \Gamma \setminus \Delta]\!]}$$

And finally, the residuation rules perform permutations of the context.

The beauty of this CPS-translation is that you can tweak which terms are interpreted as values and which as co-values. For instance, Moortgat and Moot (2011) choose to assign S *negative* polarity (co-value), and the rest of the atomic types positive polarity (value). With such an assignment, you can obtain the following results:

$$\text{john} : \cdot\text{NP}\cdot \bullet \text{leaves} : \cdot\text{IV}\cdot \vdash \cdot\text{S}\cdot$$
$$\downarrow$$
$$(\lambda k.\text{leaves } (\text{john}, k))$$

Note that the verb gets access to the top continuation. This is useful when interpreting multiple utterances.

We can take top-level scope from any function which returns someone with a positive type:

$$\text{john} : \cdot\text{NP}\cdot \bullet \text{finds} : \cdot\text{TV}\cdot \bullet \text{a} : \cdot\text{NP} \,/\, \text{N}\cdot \bullet \text{unicorn} : \cdot\text{N}\cdot \vdash \cdot\text{S}\cdot$$
$$\downarrow$$
$$(\lambda k.\text{some } ((\lambda y.\text{finds } ((\text{john}, k), y)), \text{unicorn}))$$

We get exactly the scope ambiguity we would expect in natural language:

$$\text{every} : \cdot\text{NP} \,/\, \text{N}\cdot \bullet \text{person} : \cdot\text{N}\cdot \bullet \text{loves} : \cdot\text{TV}\cdot \bullet \text{some} : \cdot\text{NP} \,/\, \text{N}\cdot \bullet \text{person} : \cdot\text{N}\cdot \vdash \cdot\text{S}\cdot$$
$$\downarrow$$
$$(\lambda k.\text{every } ((\lambda x.\text{some } ((\lambda y.\text{loves } ((x, k), y)), \text{person})), \text{person}))$$
$$(\lambda k.\text{some } ((\lambda x.\text{every } ((\lambda y.\text{loves } ((y, k), x)), \text{person})), \text{person}))$$

In this system, the scope-taking occurs when e.g. $\cdot\text{NP} \,/\, \text{N}\cdot \bullet \cdot\text{N}\cdot$ is merged into NP. The polarity assignment allows for the following derivation, which will insert the quantifier in the top scope:

$$\dfrac{\dfrac{\dfrac{\vdots}{p : \cdot\text{N}\cdot \vdash \boxed{p : \text{N}}} \quad \dfrac{x : \cdot\text{NP}\cdot \vdash M : \Gamma}{\boxed{(\lambda x.M\ y) : \text{NP}} \vdash y : \Gamma}\,\text{Unf}^L}{\boxed{(\text{case } z \text{ of } (y, p) \to (\lambda x.M\ y, p)) : \text{NP} \,/\, \text{N}} \vdash z : \Gamma \,/\, \cdot\text{N}\cdot}\,\text{L}/}{k : \cdot\text{NP} \,/\, \text{N}\cdot \vdash (\lambda z.k\ (\text{case } z \text{ of } (y, p) \to (\lambda x.M\ y, p))) : \Gamma \,/\, \cdot\text{N}\cdot}\,\text{Foc}^L$$

Because the result type is positive, we can unfocus. This causes the ambiguity: we can choose between starting out with "every person" or "some person".[17] In addition, when we add logical products to the calculus (as Moortgat and Moot do) we can write the types of "everyone" and "someone" as $(\text{NP} \,/\, \text{N}) \otimes \text{N}$, and obtain the same scope-taking behaviour.

[17] Because NLQ uses a direct translation, instead of the CPS translation, this ambiguity would be spurious ambiguity. Therefore, NLQ assigns uniform *negative* polarities, which prevents this type of ambiguity.

4.2.3 Limitations: dynamic answer types, scope islands and strong and weak quantifiers

Moortgat and Moot's (2011) CPS-semantics give a beautiful solution to the problem of spurious ambiguity both in proof search *and* in CPS-translation. However, their solution—and Barker's (2002)—still suffers from two related problems:

1. Both require a *static* choice of answer type. This means that the answer type (referred to as R above) cannot change throughout the program.
2. Both cannot encode *delimiters* past which a nested expression cannot take scope.

The first of these two is a problem when analysing sentences such as "Alice read a book [the author of which] feared the ocean". The indented interpretation of this sentence is:[18]

$$\exists x.(\mathbf{book}(x) \wedge \mathbf{fear}(\iota(\lambda y.(\mathbf{of}(x, \mathbf{author}, y))), \iota(\lambda z.\mathbf{ocean}(z)))) \wedge \mathbf{read}(\mathbf{alice}, x)$$

The system NLQ, and other, similar systems are capable of deriving these semantics, by having "which" take scope at the top of the embedded clause "the author of which". It will take this function, which has type **ee**, apply it to the existentially quantified variable, and insert the result as the subject of the gapped sentence "_ feared the ocean". As this involves a quantifier taking scope at the top of an NP node, instead of the usual S node, it is unclear how the aforementioned approaches could derive this interpretation.

The second is the problem of scope restriction. There are many different forms of scope restriction in natural language (see Szabolcsi, 2000), but one of the clearest examples to me is a phrase such as "Mary said everyone left". If we assume that "everyone", as a quantifier, can take scope at the top-level, we get the following interpretation:

$$\forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{say}(\mathbf{mary}, \mathbf{past}(\mathbf{leave}(x))))$$

The embedded clause "everyone left" is also a sentence, so everyone should also be able to take scope there, giving us the following semantics:

$$\mathbf{past}(\mathbf{say}(\mathbf{mary}, \forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{leave}(x))))$$

There is, however, something off about the first interpretation for the sentence, where everyone takes scope at the top-level. If we think about what the two logical formulas mean, the first one means that Mary made a speech act, in which she declared "everyone left". The second interpretation, however, states that for each person Mary made a *separate* speech act, in which she declared that that particular person had left.

There is more to the story of delimiters: for a sentence such as "Everyone said someone left", we expect a reading similar to the second reading above:

$$\forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{say}(x, \exists y.\mathbf{person}(x) \wedge \mathbf{past}(\mathbf{leave}(y))))$$

[18] The semantics for 'the' are commonly given in terms of a function called 'ι', known as the "definite description operator." Given a set, this operator returns its unique inhabitant. Depending on the exact semantics you want for this operator, it can either be implemented as a side-effect (e.g. using monadic semantics), or using quantification. For this example, it does not matter which solution we choose, so think of 'ι' as a function of the type (**et**)**e**.

This would mean that there is one particular person, and everyone said that that person had left. However, in this case, it does not seem absurd to also allow the other reading:

$$\forall x.\mathbf{person}(x) \supset (\exists y.\mathbf{person}(x) \wedge \mathbf{past}(\mathbf{say}(x, \mathbf{past}(\mathbf{leave}(y)))))$$

That is to say: everyone said that someone has left, but they may be talking about different people. This phenomenon is known as *strong* vs. *weak* quantifiers; the hypothesis is that quantifiers, based on their strength, may be able to scope out of a scope island. Neither Barker (2002, 2004) nor Moortgat and Moot (2011) can analyse these phenomena.

4.2.4 Continuation hierarchy and scope

Kiselyov and Shan (2014) present an elegant solution to the problem of strong and weak quantifiers: they set up a framework in which expressions can be CPS-translated repeatedly, whereby expressions that are lifted more often are stronger than expressions that are lifted fewer times. They proceed by showing that in this framework they can account for scope islands, scope ambiguity without over-generation, and *any number* of quantifier strengths.[19] They obtain both scope ambiguity and scope islands by their mechanism of quantifier strength. Scope ambiguity is obtained by varying the quantifier strengths, i.e. for "Everyone loves someone" we have "Everyone$_2$ loves someone$_1$" and "Everyone$_1$ loves someone$_2$", where the strength of the quantifier determines the scope. Scope islands are implemented by means of semantic operator, which lowers the quantifier strength of its argument by a certain amount, therefore preventing weak quantifiers from out-scoping it. It is, however, unclear whether or not this framework can handle changing answer types.

4.3 Syntactic approaches to scope

The idea of using structural postulates to treat scope is, by now, rather old. Morrill (1994), for instance, introduces the $\uparrow$ and $\downarrow$ connectives for discontinuous constituents.

Moortgat (1996) discusses two key techniques for extending logics with structural postulates, while maintaining control of where these apply: *multimodality* and *licensing*. Multimodality, in essence, means that when we add a new structural postulate—e.g. associativity for our product—we add it to a *new* copy of the connective. This way, we maintain the results we had for our logic without the structural postulate, and we keep our structural postulates from interfering. Licensing, on the other hand, means restricting access to new modalities, or new structural rules. This technique is employed, for instance, by Girard (1987) to restrict the access to contraction and weakening in full linear logic.

Another contribution in Moortgat (1996) is the description of the $q(A, B, C)$ connective—a first attempt to formalise which theorems should be derivable for a syntactic implementation of quantification. Moortgat writes:

> As a first approximation, [below we present] the connective $q(A, B, C)$ for 'in situ' binding [..]. Use of a formula $q(A, B, C)$ binds a variable

[19] We are not aware of any work in linguistics that shows that more than two strengths are necessary. Nonetheless, Kiselyov and Shan's (2014) general solution is quite nice.

x of type A, where the resource A is substituted for [..] $q(A, B, C)$ in the binding domain B. Using $q(A, B, C)$ turns the binding domain B into C. In the generalized quantifier case we have typing $q(np, s, s)$ where it happens that $B = C = s$.

$$\frac{\Delta[x : A] \vdash M : B \qquad \Gamma[y : C] \vdash N : D}{\Gamma[\Delta[z : q(A, B, C)]] \vdash N[z\ (\lambda x.M)/y] : D}\ q\mathrm{L}$$

He then proceeds to give an implementation of this connective in terms of more primitive, logical rules, by adding the following structural postulates:

$$\Diamond A \longleftrightarrow A \circ \mathbf{t} \tag{P0}$$
$$(A \circ B) \bullet C \longleftrightarrow A \circ \langle l \rangle (B \bullet C) \tag{P1}$$
$$A \bullet (B \circ C) \longleftrightarrow B \circ \langle r \rangle (A \bullet C) \tag{P2}$$

These postulates allow the system to: (1) take a formula marked with a $\Diamond$ (the quantifying license); (2) turn the $\Diamond$ it into a trace ($\mathbf{t}$), thereby unlocking a *hollow* product (the quantifying modality); and (3) iteratively move the formula upwards, leaving a trail of $\langle l \rangle$s and $\langle r \rangle$s. He defines the $q(A, B, C)$ connective as $(B \,/_w A) \setminus_w C$ (where w is, once again, the quantifying modality). Once the quantifier reaches a position where it can be reduced—for quantifiers $q(np, s, s)$ this is usually the top—we can reduce it using $\mathrm{L}/_w$ and $\mathrm{R}\setminus_w$.

Note that contrary to most linguistic frameworks, in which quantifier movement conceived of as leaving a trace, and moving is strictly upwards (i.e. quantifier raising), in this framework it ends up being characterised by upwards movement followed by downwards movement *along the same path*—i.e. scope-taking followed by the insertion of the newly bound variable.

As the derivations given in Moortgat (1996) are almost identical to contemporary derivations given using $\mathrm{NL}_{\mathrm{IBC}}$ and NLQ, and both will be discussed in later sections, we will refrain from presenting such a derivation here.

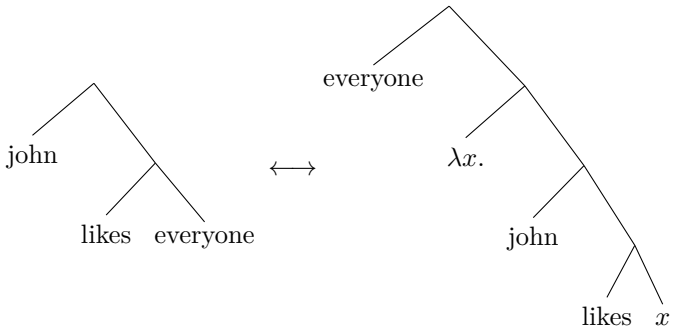
4.3.1 $\mathbf{NL}_\lambda$, $\mathbf{NL_{CL}}$ and $\mathbf{NL_{IBC}}$

Barker (2007) and Barker and Shan (2014) describe an extension to NL which they call NL_λ (sometimes $\mathrm{NL}_{\mathrm{QR}}$). Its main contributions over Moortgat (1996) are the introduction of *parasitic scope*—a syntactic mechanism to capture the variables bound by another quantifier—and a related refinement of Moortgat's (1996) $q(A, B, C)$ and $q\mathrm{L}$. This contributions—as paraphrased by us to fit Moortgat's (1996) notation—consists of a binary connective $q(A, B)$, and a deconstruction of the old $q\mathrm{L}$ into $q\mathrm{L}$ and $q\mathrm{R}$ (where Σ' is some variation of Σ):

$$\frac{\Sigma' \vdash q(A, B) \qquad \Gamma[C] \vdash \Delta}{\Gamma[\Sigma[q(A, B, C)]] \vdash \Delta}\ q\mathrm{L} \qquad \frac{\Sigma[A] \vdash B}{\Sigma' \vdash q(A, B)}\ q\mathrm{R}$$

We will discuss parasitic scope, and the motivation for this deconstruction, more extensively in the context of NLQ (section 4.3.3), but first we will introduce NL_λ.

The basis for the way in which NL_λ represents quantifier movement is the following tree transformation, which is often used to represent quantifier raising:



Barker and Shan implement this transformation informally by extending the syntax for structures with a binding construct $\lambda x.(\dots x \dots)$ and a new modality $\{\mathbin{\backslash\!\backslash}, \circ, \mathbin{/\!\!/}\}$, and adopting the following structural postulate:

$$\Sigma[\Delta] \longleftrightarrow \Delta \circ \lambda x.\Sigma[x] \tag{λ}$$

This results in an ad-hoc, but intuitive implementation of the $q(A, B, C)$ connective: it encodes (at least) the upward movement that Moortgat encodes with (P1) and (P2), with the inserted variable serving as the trace inserted by (P0).

The implementation feels intuitive because, if you read the structural products as function applications, (λ) is reminiscent of (linear) lambda abstraction. If we "β-reduce", the extracted structure is passed back to the function, and placed back where it was extracted from. For some reason, though—perhaps to ensure compatibility with Moortgat (1996), or with the notion of quantifier-raising to the top-left node?—Barker (2007) chooses to place the extracted structure to the left of the product, making it a flipped function application.

In NL_λ, $q(A, B) = A \mathbin{\backslash\!\backslash} B$, and $q(A, B, C) = C \mathbin{/\!\!/} q(A, B)$. This way, qL and qR can be informally implemented in display NL extended with (λ) and $\{\mathbin{\backslash\!\backslash}, \circ, \mathbin{/\!\!/}\}$ as follows:[20]

$$\dfrac{\dfrac{\dfrac{\lambda x.\Sigma[x] \vdash \cdot A \mathbin{\backslash\!\backslash} B\cdot \qquad \cdot C\cdot \vdash \Delta}{\cdot C \mathbin{/\!\!/} (A \mathbin{\backslash\!\backslash} B)\cdot \vdash \Delta \mathbin{/\!\!/} \lambda x.\Sigma[x]}\,\mathrm{L}/\!\!/}{\cdot C \mathbin{/\!\!/} (A \mathbin{\backslash\!\backslash} B)\cdot \circ \lambda x.\Sigma[x] \vdash \Delta}\,\mathrm{Res}/\!\!/\circ}{\Sigma[\cdot C \mathbin{/\!\!/} (A \mathbin{\backslash\!\backslash} B)\cdot] \vdash \Delta}\,(\lambda) \qquad \dfrac{\dfrac{\dfrac{\Sigma[\cdot A\cdot] \vdash \cdot B\cdot}{\cdot A\cdot \circ \lambda x.\Sigma[x] \vdash \cdot B\cdot}\,(\lambda)}{\lambda x.\Sigma[x] \vdash \cdot A\cdot \mathbin{\backslash\!\backslash} \cdot B\cdot}\,\mathrm{Res}\circ\mathbin{\backslash\!\backslash}}{\lambda x.\Sigma[x] \vdash \cdot A \mathbin{\backslash\!\backslash} B\cdot}\,\mathrm{R}\mathbin{\backslash\!\backslash}$$

As far as exactness is concerned, however, (λ) admits much more than just qL and qR. And some of it is… odd. For instance, since contexts are defined as structures with a hole, we can raise two quantifiers past one another, indefinitely:

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \circ \lambda z.(\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \circ \lambda y.(z \circ \lambda x.(y \bullet \cdot (\mathrm{NP} \mathbin{\backslash} \mathrm{S}) \mathbin{/} \mathrm{NP}\cdot \bullet x))) \vdash \cdot \mathrm{S}\cdot}\,\lambda}{\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \circ \lambda y.(\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \circ \lambda x.(y \bullet \cdot (\mathrm{NP} \mathbin{\backslash} \mathrm{S}) \mathbin{/} \mathrm{NP}\cdot \bullet x)) \vdash \cdot \mathrm{S}\cdot}\,\lambda}{\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \circ \lambda x.(\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \bullet \cdot (\mathrm{NP} \mathbin{\backslash} \mathrm{S}) \mathbin{/} \mathrm{NP}\cdot \bullet x) \vdash \cdot \mathrm{S}\cdot}\,\lambda}{\cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \bullet \cdot (\mathrm{NP} \mathbin{\backslash} \mathrm{S}) \mathbin{/} \mathrm{NP}\cdot \bullet \cdot \mathrm{S} \mathbin{/} (\mathrm{NP} \mathbin{\backslash} \mathrm{S})\cdot \vdash \cdot \mathrm{S}\cdot}$$

Or, as Barker and Shan (2014) note, we could lift variables out of the scope of their binder:

[20] Note that we can drop the outermost context Γ from the specification in display NL, due to the display property.

$$\dfrac{\dfrac{\dfrac{\vdots}{x \circ \lambda y.(\cdot \mathrm{S} \,/\, (\mathrm{NP} \setminus \mathrm{S}) \cdot \circ \lambda x.(\cdot \mathrm{S} \,/\, (\mathrm{NP} \setminus \mathrm{S}) \cdot \bullet \cdot (\mathrm{NP} \setminus \mathrm{S}) \,/\, \mathrm{NP} \cdot \bullet\, y)) \vdash \cdot \mathrm{S} \cdot}\,\lambda}{\cdot \mathrm{S} \,/\, (\mathrm{NP} \setminus \mathrm{S}) \cdot \circ \lambda x.(\cdot \mathrm{S} \,/\, (\mathrm{NP} \setminus \mathrm{S}) \cdot \bullet \cdot (\mathrm{NP} \setminus \mathrm{S}) \,/\, \mathrm{NP} \cdot \bullet\, x) \vdash \cdot \mathrm{S} \cdot}\,\lambda}{\cdot \mathrm{S} \,/\, (\mathrm{NP} \setminus \mathrm{S}) \cdot \bullet \cdot (\mathrm{NP} \setminus \mathrm{S}) \,/\, \mathrm{NP} \cdot \bullet \cdot \mathrm{S} \,/\, (\mathrm{NP} \setminus \mathrm{S}) \cdot \vdash \cdot \mathrm{S} \cdot}$$

While none of this is necessarily problematic for the logical properties of NL_λ, as one can probably show that none of these gimmicky derivations will ever lead to a valid proof, it is *devastating* for naive algorithms for proof search, and needlessly complicates meta-logical proofs such as the completeness of proof search, or cut-elimination. In addition, the use of a binding construct in the syntax for structures makes the system very difficult to formalise or reason about—something which perhaps shows in the nonspecific terms in which the system is presented.

In refining NL_λ, Barker and Shan create $\mathrm{NL}_{\mathrm{CL}}$ (often $\mathrm{NL}_{\mathrm{IBC}}$). The idea of this system is to deconstruct the complex structure of the linear lambda by encoding the combinators **I**, **B** and **C**, which make up the linear lambda calculus. This manner of encoding combinator calculi—including **IBC**—was extensively studied by Finger (1998). For their version, Barker and Shan keep the new modality $\{\backslash\!\backslash, \circ, /\!/\}$, and further extend this system with three structural constants, $\{\mathbf{I}, \mathbf{B}, \mathbf{C}\}$, and the following structural postulates:

$$A \longleftrightarrow A \circ \mathbf{I} \tag{\textbf{I}}$$
$$(X \circ Y) \bullet Z \longleftrightarrow X \circ ((\mathbf{C} \bullet Y) \bullet Z) \tag{\textbf{C}}$$
$$X \bullet (Y \circ Z) \longleftrightarrow Y \circ ((\mathbf{B} \bullet X) \bullet Z) \tag{\textbf{B}}$$

Quirky flipped-application notation aside, these postulates encode exactly the reduction behaviours of the combinators **I**, **B** and **C**:

$$\begin{aligned} \mathbf{I}x &\equiv x \\ \mathbf{B}xyz &\equiv x(yz) \\ \mathbf{C}xyz &\equiv xzy \end{aligned}$$

In the resulting calculus, quantifier raising can be done in much the same way as in NL_λ—though the new version is ever so slightly more verbose:[21]

[21] Inverted applications of the **I**, **B** and **C** rules are marked with a prime.

$$
\begin{array}{cl}
\vdots & \\
\cdot NP\cdot \bullet \cdot(NP \backslash S) / NP\cdot \bullet \cdot NP\cdot \vdash \cdot S\cdot & \\
\hline
\cdot(NP \backslash S) / NP\cdot \bullet \cdot NP\cdot \vdash \cdot NP\cdot \backslash \cdot S\cdot & \text{Res}\bullet\backslash \\
\hline
\cdot NP\cdot \vdash \cdot(NP \backslash S) / NP\cdot \backslash \cdot NP\cdot \backslash \cdot S\cdot & \text{Res}\bullet\backslash \\
\hline
\cdot NP\cdot \circ \mathbf{I} \vdash \cdot(NP \backslash S) / NP\cdot \backslash \cdot NP\cdot \backslash \cdot S\cdot & \mathbf{I} \\
\hline
\cdot(NP \backslash S) / NP\cdot \bullet (\cdot NP\cdot \circ \mathbf{I}) \vdash \cdot NP\cdot \backslash \cdot S\cdot & \text{Res}\backslash\bullet \\
\hline
\cdot NP\cdot \bullet ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I}) \vdash \cdot NP\cdot \backslash \cdot S\cdot & \mathbf{B} \\
\hline
\cdot NP\cdot \bullet (\cdot NP\cdot \circ ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) \vdash \cdot S\cdot & \text{Res}\bullet\backslash \\
\hline
\cdot NP\cdot \circ ((\mathbf{B} \bullet \cdot NP\cdot) \bullet ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) \vdash \cdot S\cdot & \mathbf{B} \\
\hline
((\mathbf{B} \bullet \cdot NP\cdot) \bullet ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) \vdash \cdot NP\cdot \backslash\!\backslash \cdot S\cdot & \text{Res}\circ\backslash\!\backslash \\
\hline
((\mathbf{B} \bullet \cdot NP\cdot) \bullet ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) \vdash \cdot NP \backslash\!\backslash S\cdot \qquad \dfrac{}{\cdot S\cdot \vdash \cdot S\cdot}\,\text{Ax} & \text{R}\backslash\!\backslash \\
\hline
\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \vdash \cdot S\cdot /\!/ ((\mathbf{B} \bullet \cdot NP\cdot) \bullet ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) & \text{L}/\!/ \\
\hline
\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \circ ((\mathbf{B} \bullet \cdot NP\cdot) \bullet ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) \vdash \cdot S\cdot & \text{Res}/\!/\circ \\
\hline
\cdot NP\cdot \bullet (\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \circ ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I})) \vdash \cdot S\cdot & \mathbf{B}' \\
\hline
\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \circ ((\mathbf{B} \bullet \cdot(NP \backslash S) / NP\cdot) \bullet \mathbf{I}) \vdash \cdot NP\cdot \backslash \cdot S\cdot & \text{Res}\bullet\backslash \\
\hline
\cdot(NP \backslash S) / NP\cdot \bullet (\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \circ \mathbf{I}) \vdash \cdot NP\cdot \backslash \cdot S\cdot & \mathbf{B}' \\
\hline
\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \circ \mathbf{I} \vdash \cdot(NP \backslash S) / NP\cdot \backslash \cdot NP\cdot \backslash \cdot S\cdot & \text{Res}\backslash\bullet \\
\hline
\cdot S /\!/ (NP \backslash\!\backslash S)\cdot \vdash \cdot(NP \backslash S) / NP\cdot \backslash \cdot NP\cdot \backslash \cdot S\cdot & \mathbf{I}' \\
\hline
\cdot(NP \backslash S) / NP\cdot \bullet \cdot S /\!/ (NP \backslash\!\backslash S)\cdot \vdash \cdot NP\cdot \backslash \cdot S\cdot & \text{Res}\bullet\backslash \\
\hline
\cdot NP\cdot \bullet \cdot(NP \backslash S) / NP\cdot \bullet \cdot S /\!/ (NP \backslash\!\backslash S)\cdot \vdash \cdot S\cdot & \text{Res}\bullet\backslash
\end{array}
$$

One of the advantages of this formalisation is that it gets rid of the awkward binding construct in the syntax of structures. In addition, it formalises the intuition that quantifiers can only move past *solid* products. However, it is not entirely free of problems. One of the more glaring problems is that using this encoding, any expression can be the subject of quantifier raising. For instance, in "John likes Mary," we could choose to raise the verb:

$$
\begin{array}{c}
\vdots \\
\cdot(NP \backslash S) / NP\cdot \circ (\mathbf{B} \bullet \cdot NP\cdot) \bullet ((\mathbf{C} \bullet \mathbf{I}) \bullet \cdot NP\cdot) \vdash \cdot S\cdot \\
\vdots \\
\cdot NP\cdot \bullet \cdot(NP \backslash S) / NP\cdot \bullet \cdot NP\cdot \vdash \cdot S\cdot
\end{array}
$$

Since verbs are usually not considered scope-takers, it is unlikely that raising the verb would lead to anything other then having to lower it again. However, the fact that we leave it open as an opportunity is wasted computational effort; while all futile attempts at raising and lowering will lead to a loop, and therefore spare us the spurious ambiguity, there are still a great deal of futile attempts to be made.

Another problem is the **I**-rule. It allows us to introduce an arbitrary amount of **I**'s, which causes a *growing* loop in our proof search procedure:

$$
\begin{array}{cl}
\vdots & \\
((\cdot NP\cdot \bullet \cdot NP \backslash S\cdot) \circ \mathbf{I}) \circ \mathbf{I} \vdash \cdot S\cdot & \\
\hline
(\cdot NP\cdot \bullet \cdot NP \backslash S\cdot) \circ \mathbf{I} \vdash \cdot S\cdot & \mathbf{I}' \\
\hline
\cdot NP\cdot \bullet \cdot NP \backslash S\cdot \vdash \cdot S\cdot & \mathbf{I}'
\end{array}
$$

This means that $\mathrm{NL}_{\mathrm{CL}}$, while closer, is still undecidable. In the following section, we will present the first extension for NLQ, and address this issue of decidability.

4.3.2 IBC for NLQ

In this section we will present a set of structural postulates, based on Barker and Shan's (2014) $\mathrm{NL}_{\mathrm{CL}}$, for which proof search is decidable. But first, we will remove the quirky reversed nature of the hollow product ($\circ$), and make the connection with combinator calculus explicit:[22]

$$\frac{X \vdash Y}{\mathbf{I} \circ X \vdash Y}\,\mathbf{I} \qquad \frac{X \bullet (Y \circ Z) \vdash W}{((\mathbf{B} \bullet X) \bullet Y) \circ Z \vdash W}\,\mathbf{B} \qquad \frac{(X \circ Z) \bullet Y \vdash W}{((\mathbf{C} \bullet X) \bullet Y) \circ Z \vdash W}\,\mathbf{C}$$

Now, in the previous section we mentioned that $\mathrm{NL}_{\mathrm{IBC}}$ has two major flaws: it does not license quantifier raising, and it is undecidable. We propose to handle both of these issues with one simple addition. The idea is to add a new unary connective, $\mathbf{Q}(A)$, which represents a *license* to perform quantifier raising. Since we want to replace the problematic **I**-rule, we will choose the structural version of our quantifying license to be a *hollow* product with unit as its left-hand argument. On the other side, since we do not necessarily want logical products, we will keep $\mathbf{Q}(A)$ it as an atomic logical connective. This gives us the following logical left and right rules:

$$\frac{\mathbf{I} \circ \cdot A \cdot \vdash \Delta}{\cdot \mathbf{Q}(A) \cdot \vdash \Delta}\,\mathrm{L}\mathbf{I} \qquad \frac{\Gamma \vdash \cdot B \cdot}{\mathbf{I} \circ \Gamma \vdash \cdot \mathbf{Q}(B) \cdot}\,\mathrm{R}\mathbf{I}$$

And indeed, the pair obeys all constraints imposed by display calculus, including a valid proof for **C8**:

$$\dfrac{\dfrac{\Gamma \vdash \cdot A \cdot \qquad \dfrac{\mathbf{I} \circ \cdot A \cdot \vdash \Delta}{\cdot A \cdot \vdash \mathbf{I} \mathbin{\backslash\!\!\backslash} \Delta}\,\mathrm{Res}\circ/\!/}{\Gamma \vdash \mathbf{I} \mathbin{\backslash\!\!\backslash} \Delta}\,\mathrm{Cut}}{\mathbf{I} \circ \Gamma \vdash \Delta}\,\mathrm{Res}/\!/\circ$$

Note that we must keep the **I**-rule, though we rename it $\mathbf{I}^{-}$ to emphasise that it can now only *remove* **I**s. The full extension, including semantics, and focused rules, can be found in figure 12. The semantics are rather trivial: we simply translate all constants as units, and translate $\mathbf{Q}(A)$ as A, inserting and removing the left unit as needed.

What remains now is to show that we have indeed implemented qL and qR. For this, we will need the following definitions:

$$\text{Context } \Sigma \coloneqq \Box \mid \Sigma \bullet \Gamma \mid \Gamma \bullet \Sigma$$

$$\begin{array}{lll} \Box & [\Gamma'] \mapsto \Gamma' & \overline{\Box} \quad \mapsto \mathbf{I} \\ (\Sigma \bullet \Gamma) & [\Gamma'] \mapsto (\Sigma[\Gamma'] \bullet \Gamma) & \overline{\Sigma \bullet \Gamma} \mapsto ((\mathbf{C} \bullet \overline{\Sigma}) \bullet \Gamma) \\ (\Gamma \bullet \Sigma) & [\Gamma'] \mapsto (\Gamma \bullet \Sigma[\Gamma']) & \overline{\Gamma \bullet \Sigma} \mapsto ((\mathbf{B} \bullet \Gamma) \bullet \overline{\Sigma}) \end{array}$$

[22] The downside of this is that we raise quantifiers to the top-right node instead of the top-left—something which makes no difference, but may feel unintuitive to people who are used to reading from left-to-right.

First we have contexts, which encode structures with a single hole, where the nodes leading up to the hole are all solid products—exactly the type of structure that quantifiers can move up through. The syntax is a little abusive, as we use the same symbol for products, products with a hole in their left argument, and products with a hole in their right argument. However, as we require that contexts have only a single hole, it is always unambiguous. Note that products are right-associative. Secondly, we have the plugging function ' $\cdot[\cdot]$', which inserts some structure into the single hole in a context. Lastly, the 'trace' function computes, from a context, the trace of **B**s and **C**s that would be left after something quantifies out of that context.

Given these definitions, we can show that the following rule for quantifier movement is admissible, by induction on the structure of the context:

$$\frac{\overline{\Sigma} \circ \cdot A \cdot \vdash \Delta}{\Sigma[\cdot \mathbf{Q}(A) \cdot] \vdash \Delta} \uparrow\downarrow$$

Note that this new rule is very close to Barker and Shan's (λ), with as its only difference that their version is flipped, and ours has a quantifying license built into it:

$$\Sigma[\Delta] \longleftrightarrow \Delta \circ \lambda x.\Sigma[x] \qquad \Sigma[\cdot \mathbf{Q}(A) \cdot] \longleftrightarrow \overline{\Sigma} \circ \cdot A \cdot$$

Proof search with this derived rule is still complete; it merely enforces that the entire quantifier raising or lowering is done in a single movement, as the quantifying license is consumed by the application. Interestingly, ↑↓ also has the sub-formula property. This means that ↑↓ can serve as a normal-form for **LI**, R**I**, $\mathbf{I}^-$, **B** *and* **C**, and that proof search with ↑↓ is both complete *and* decidable!

Onwards: in their discussion of decidability, Barker and Shan (2014) derive 'expansion' and 'reduction' rules which more-or-less correspond to the two directions of (λ). They then combine these rules with L⫽ and R⧵, yielding qL and qR (which they call ⫽L$_\lambda$ and ⧵R$_\lambda$). We can do the same, using ↑↓:

$$\frac{\overline{\Sigma} \vdash \cdot B \mathbin{/\!\!/} A \cdot \qquad \cdot C \cdot \vdash \Delta}{\Sigma[\cdot \mathbf{Q}((B \mathbin{/\!\!/} A) \mathbin{\backslash\!\!\backslash} C) \cdot] \vdash \Delta}\, q\mathrm{L} \qquad \frac{\Sigma[\cdot A \cdot] \vdash \cdot B \cdot}{\overline{\Sigma} \vdash \cdot B \mathbin{/\!\!/} A \cdot}\, q\mathrm{R}$$

If we permit ourselves the assumption that all quantifiers in natural language can be described by $q(A, B, C)$—not an unreasonable assumption, albeit not one we enforce—then qL and qR too can serve as normal-forms. And, to boot, we use them to write proofs involving quantifier movement in a much more succinct manner:

$$\dfrac{\dfrac{\dfrac{\vdots}{\cdot \mathrm{NP} \cdot \bullet \textsc{loves} \bullet \cdot \mathrm{NP} \cdot \vdash \cdot \mathrm{S} \cdot}}{\dfrac{\overline{\square \bullet \textsc{loves} \bullet \cdot \overline{\mathrm{NP}} \cdot} \vdash \cdot \mathrm{S} \mathbin{/\!\!/} \mathrm{NP} \cdot}\, q\mathrm{R} \qquad \dfrac{}{\cdot \mathrm{S} \cdot \vdash \cdot \mathrm{S} \cdot}\, \mathrm{Ax}}{\textsc{everyone} \bullet \textsc{loves} \bullet \cdot \mathrm{NP} \cdot \vdash \cdot \mathrm{S} \cdot}\, q\mathrm{L}}{\dfrac{\overline{\textsc{everyone} \bullet \textsc{loves} \bullet \square} \vdash \cdot \mathrm{S} \mathbin{/\!\!/} \mathrm{NP} \cdot}\, q\mathrm{R} \qquad \dfrac{}{\cdot \mathrm{S} \cdot \vdash \cdot \mathrm{S} \cdot}\, \mathrm{Ax}}{\textsc{everyone} \bullet \textsc{loves} \bullet \textsc{someone} \vdash \cdot \mathrm{S} \cdot}\, q\mathrm{L}$$

$$\downarrow$$

$$(\text{someone } (\lambda y.\text{everyone } (\lambda x.\text{likes } y\ x)))$$

$$\downarrow$$

$$\exists y.\mathbf{person}(y) \wedge \forall x.\mathbf{person}(x) \supset \mathbf{like}(x, y)$$

Note that though we have the option, we do not have to unfold the application of 'trace' in this particular proof. In future proofs, if we choose to fold or unfold an application of 'trace', we will explicitly mark this as an application of rewriting by equality ('=').

In its current shape, NLQ is capable of analysing sentences with changing answer types—something which is problematic to many semantic approaches to quantifier raising (see section 4.2.3)

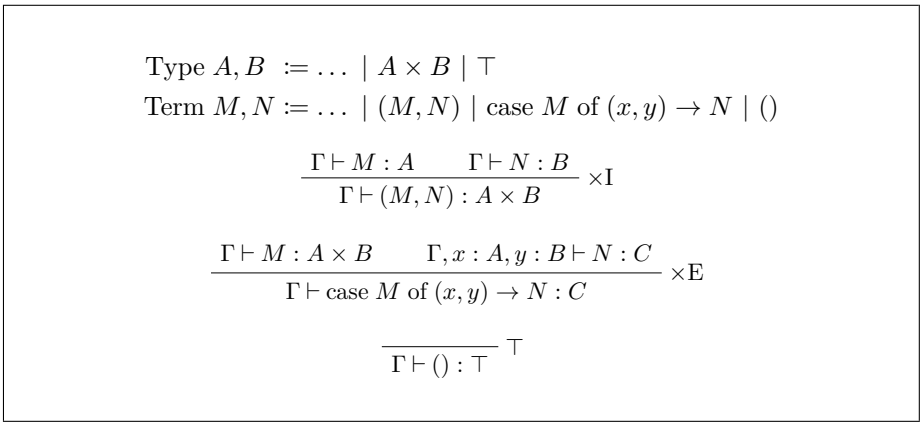


Figure 8: An extension of the semantic calculus from figure 2.

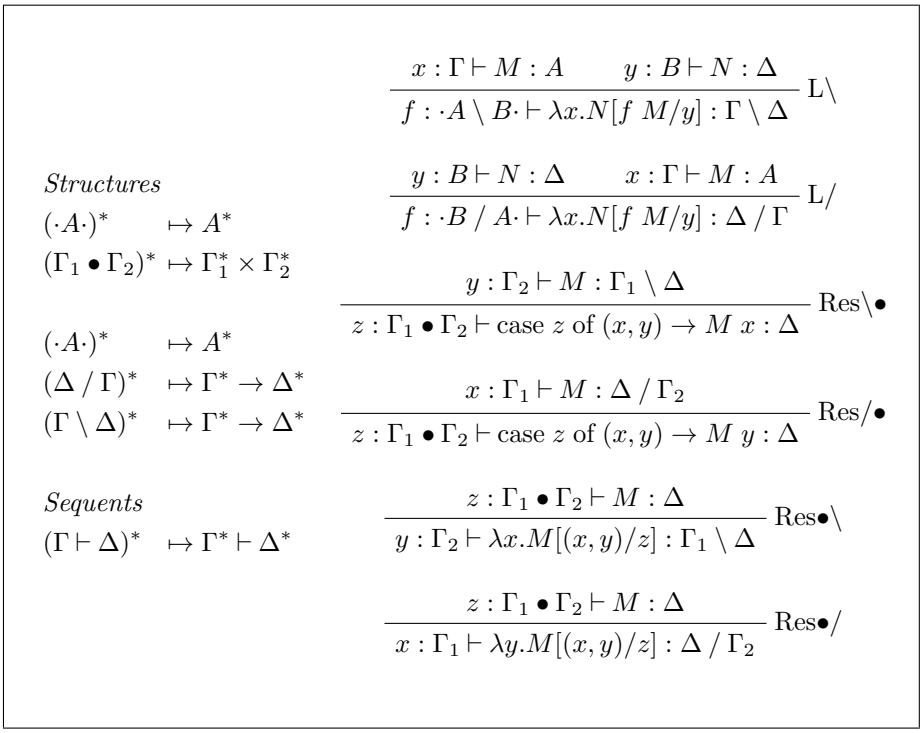


Figure 9: Term labelling for display NL.

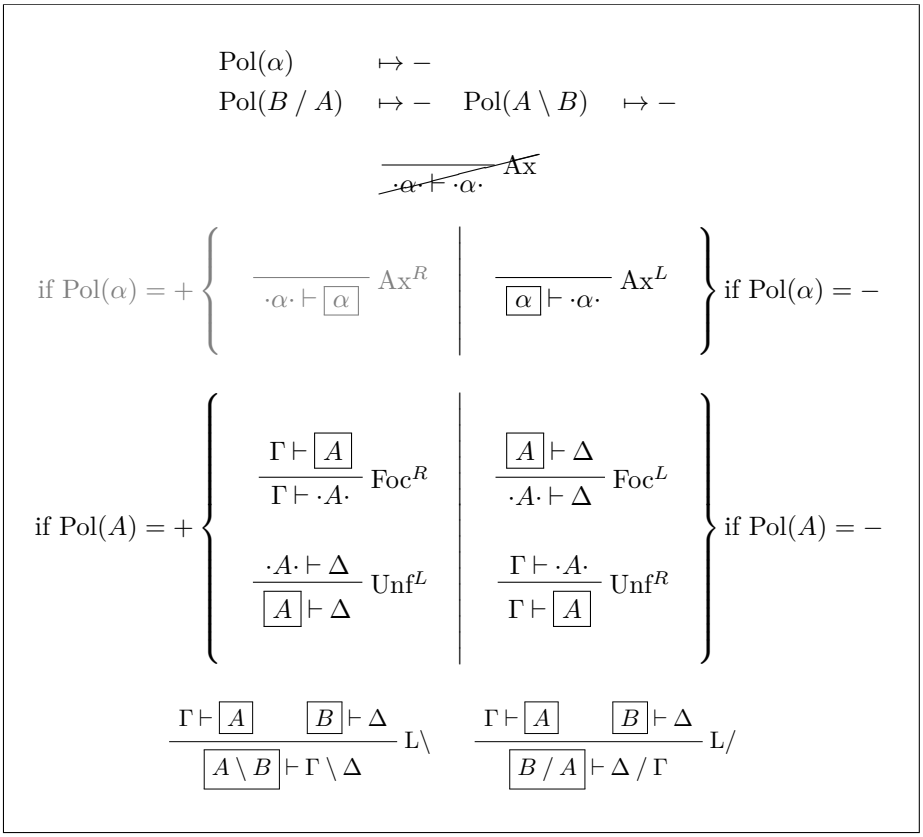


Figure 10: Changes to the display calculus from figure 7, implementing focusing.

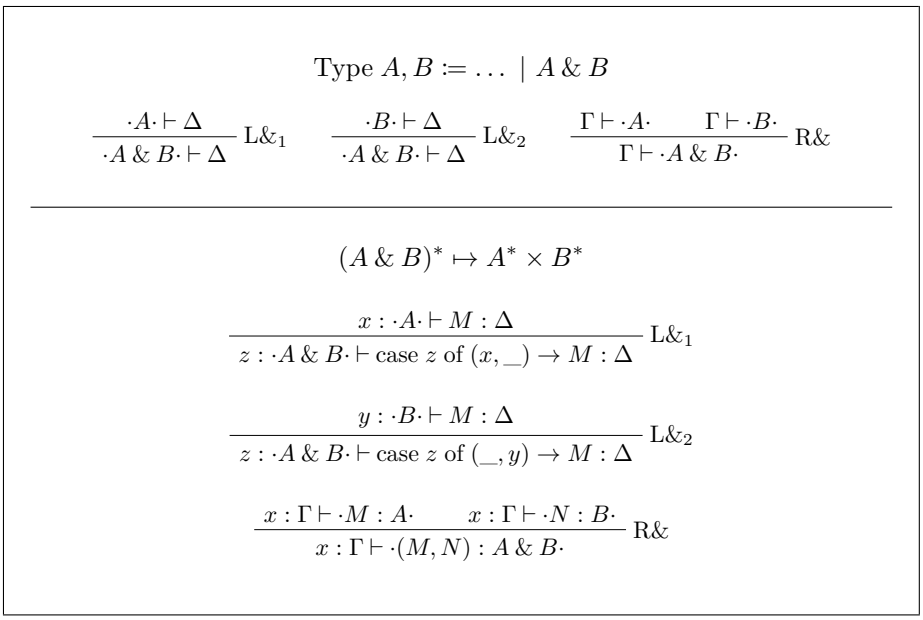


Figure 11: Extension of calculus in figure 7 which supports ambiguity.

$$\begin{array}{rll} \text{Type} & A, B \coloneqq \ldots \mid A \mathbin{\backslash\!\backslash} B \mid B \mathbin{/\!/} A \mid \mathbf{Q}(A) & \mathrm{Pol}(A \mathbin{\backslash\!\backslash} B) \mapsto - \\ \text{Structure}^+ & \Gamma \coloneqq \ldots \mid \Gamma_1 \circ \Gamma_2 \mid \mathbf{I} \mid \mathbf{B} \mid \mathbf{C} & \mathrm{Pol}(B \mathbin{/\!/} A) \mapsto - \\ \text{Structure}^- & \Delta \coloneqq \ldots \mid \Gamma \mathbin{\backslash\!\backslash} \Delta \mid \Delta \mathbin{/\!/} \Gamma & \mathrm{Pol}(\mathbf{Q}(A)) \mapsto + \end{array}$$

(copy of rules for $\{\backslash, \bullet, /\}$ from figure 7 for $\{\mathbin{\backslash\!\backslash}, \circ, \mathbin{/\!/}\}$)

$$\frac{\mathbf{I} \circ \cdot A \cdot \vdash \Delta}{\cdot \mathbf{Q}(A) \cdot \vdash \Delta}\,\mathrm{L}\mathbf{I} \qquad \frac{\Gamma \vdash \boxed{B}}{\mathbf{I} \circ \Gamma \vdash \boxed{\mathbf{Q}(B)}}\,\mathrm{R}\mathbf{I} \qquad \frac{\Gamma \vdash \Delta}{\mathbf{I} \circ \Gamma \vdash \Delta}\,\mathbf{I}^-$$

$$\frac{\Gamma_1 \bullet (\Gamma_2 \circ \Gamma_3) \vdash \Delta}{((\mathbf{B} \bullet \Gamma_1) \bullet \Gamma_2) \circ \Gamma_3 \vdash \Delta}\,\mathbf{B} \qquad \frac{(\Gamma_1 \circ \Gamma_3) \bullet \Gamma_2 \vdash \Delta}{((\mathbf{C} \bullet \Gamma_1) \bullet \Gamma_2) \circ \Gamma_3 \vdash \Delta}\,\mathbf{C}$$

$$(\mathbf{Q}(A))^* \mapsto A^*$$

$$\mathbf{I}^* \mapsto \top \quad \mathbf{B}^* \mapsto \top \quad \mathbf{C}^* \mapsto \top$$

(copy of translations for $\{\backslash, \bullet, /\}$ from figure 9 for $\{\mathbin{\backslash\!\backslash}, \circ, \mathbin{/\!/}\}$)

$$\frac{x : \mathbf{I} \circ \cdot A \cdot \vdash M : \Delta}{y : \cdot \mathbf{Q}(A) \cdot \vdash M[((), y)/x] : \Delta}\,\mathrm{L}\mathbf{I} \qquad \frac{x : \Gamma \vdash \boxed{M : B}}{y : \mathbf{I} \circ \Gamma \vdash \boxed{M[\mathrm{snd}\ y/x] : \mathbf{Q}(B)}}\,\mathrm{R}\mathbf{I}$$

$$\frac{x : \Gamma \vdash M : \Delta}{y : \mathbf{I} \circ \Gamma \vdash M[\mathrm{snd}\ y/x] : \Delta}\,\mathbf{I}^-$$

(where $\mathrm{snd}\ p = \mathrm{case}\ p\ \mathrm{of}\ (x, y) \to y$)

(**B** and **C** translate to various combinations of associativity, commutativity, $\varnothing$E and weakening)

Figure 12: Extension of calculus in figure 7 which supports quantifier raising.

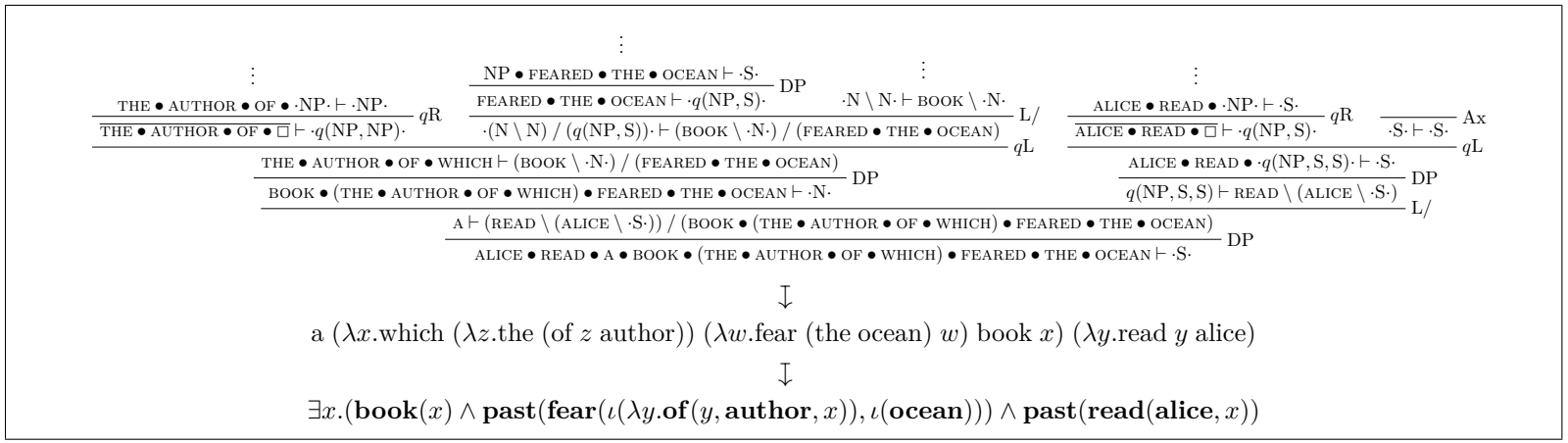


Figure 13: An example of changing answer types.

4.3.3 Parasitic scope, double quantifiers

Parasitic scope is a interesting mechanism, put forward by Barker (2007), which allows expressions to capture another quantifier's scope. Using this mechanism, a parasitic expression can take scope *right* underneath another quantifier, and capture the bound variable. For instance, a parasitic expression k might take scope under the bounded quantifier 'everyone', receiving both its context and the bound variable x as arguments:

$$\forall x.\mathbf{person}(x) \supset (k \dots x)$$

The mechanism depends crucially on two properties:[23] (1) quantification must be (at least) a two-step process; and (2) the 'function' that becomes the argument of the quantifier must be represented in the type. Both NLQ and NL_λ respects these requirements: we have $\text{q}L$ and $\text{q}R$, and the type of the 'function' that is quantifier-over is represented $q(A, B)$, which is implemented as a function-type in both cases. Using this, we can write a quantifier which targets the type of raised quantifiers. For instance,

$$q(\dots, q(\text{NP}, \text{S}), q(\text{NP}, \text{S}))$$

In NLQ, this is is implemented by the following type:

$$\mathbf{Q}(((\text{S} \mathbin{/\!\!/} \text{NP}) \mathbin{/\!\!/} \dots) \mathbin{\backslash\!\!\backslash} (\text{S} \mathbin{/\!\!/} \text{NP}))$$

Canonical examples of such quantifiers are expressions such as 'same' and 'different', as used in the sentence "The same waiter served everyone", which Barker (2007) assigns more-or-less the following semantics:[24]

$$\text{everyone}\ (\lambda X.\exists f_{(\mathbf{et})\mathbf{e}}.\forall x < X : \mathbf{past}(\mathbf{serve}(x, \iota(f(\mathbf{waiter})))))$$

Barker describes this as meaning that "[e]veryone collectively has the property of being a group such that there is a unique waiter who served each member of the group". It is clear that the intention is to have x range over the same set of variables over which 'everyone' ranges. However, it is not entirely clear to me how 'everyone' should reduce—it is ostensibly not a quantifier, but a function which takes a continuation and provides it with the set of all people (e.g. everyone $= \lambda k.k$ **person**). All this seems to be a whole lot of work to push the existential selecting the choice function up over the quantifier introduced by 'everyone'.

Kiselyov (2015) provides much clearer semantics for the semantics of 'same' and 'different' sentences:

$$\exists z.\forall y.\mathbf{past}(\mathbf{serve}(\iota(\lambda x.\mathbf{waiter}(x) \wedge x = z), y))$$

The crucial point seems to be that words such as 'same' want to take scope *over* another quantifier, but in the meantime also want access to the variable bound

[23] Though I do not rule out the possibility that a similar the mechanism can exists under different circumstances.

[24] I suppose that selection of the choice function f should be bounded by a predicate such as $\exists z.\forall x.f(x) = z$, as would be dictated by the semantics of same.

by that quantifier. In order to derive these semantics, we use a different type for parasitic quantifiers—a double quantifier:

$$q(q(\ldots, q(\mathrm{NP}, \mathrm{S}), q(\mathrm{NP}, \mathrm{S})), \mathrm{S}, \mathrm{S})$$

In NLQ, this is implemented by the following type:

$$\mathbf{Q}((\mathrm{S} /\!\!/ \quad \mathbf{Q}(((\mathrm{S} /\!\!/ \mathrm{NP}) /\!\!/ \quad \ldots \quad) \setminus\!\!\setminus (\mathrm{S} /\!\!/ \mathrm{NP}))) \quad \setminus\!\!\setminus \mathrm{S})$$

Such double quantification has some interesting aspects. First of all, one might expect huge amounts of quantifier ambiguity. However, because a double quantifier has to take scope once *normally* and once *parasitically*, in the case where there is one double and one normal quantifier, there is not room for ambiguity. The double quantifier *has* to take scope first, in order to be able to take scope parasitically while the other quantifier is taking scope.

In figure 15, we give an analysis of the sentence "A different waiter served everyone", for which we derive the following semantics, also assigned by Kiselyov:

$$\begin{aligned}&\exists f_{\mathbf{eet}}.(\forall x.\forall y.\nexists z.f\ z\ x \wedge f\ z\ y) \wedge \\ &\quad (\forall y.\mathbf{person}(y) \supset (\exists x.\mathbf{waiter}(x) \wedge f\ y\ x \wedge \mathbf{past}(\mathbf{serve}(y, x))))\end{aligned}$$

The first clause, $\forall x.\forall y.\nexists z.f\ z\ x \wedge f\ z\ y$, is entirely contained withing the semantics for 'different'—it enforces that f does not assign the same output z to two different inputs. The full lexicon used in the derivation is given in figure 14

a	:	$(q(\mathrm{NP}, \mathrm{S}, \mathrm{S}) \,/\, \mathrm{N})^*$
	=	$\lambda n.\lambda k.\exists x.n\ x \wedge k\ x$
different	:	$(q(q(\mathrm{A}, q(\mathrm{NP}, \mathrm{S}), q(\mathrm{NP}, \mathrm{S})), \mathrm{S}, \mathrm{S}))^*$
	=	$\lambda k.\exists f.(\forall x.\forall y.\nexists z.f\ z\ x \wedge f\ z\ y) \wedge$
		$k\ (\lambda k'.\lambda x.k'\ (\lambda g.\lambda y.g\ y \wedge f\ x\ y)\ x)$
waiter	:	N^*
	=	$\lambda x.\mathbf{waiter}(x)$
served	:	TV^*
	=	$\lambda y.\lambda x.\mathbf{past}(\mathbf{serve}(x, y))$
everyone	:	$(q(\mathrm{NP}, \mathrm{S}, \mathrm{S}))^*$
	=	$\lambda k.\forall x.\mathbf{person}(x) \supset k\ x$

Figure 14: Lexicon used in the derivation in figure 15.

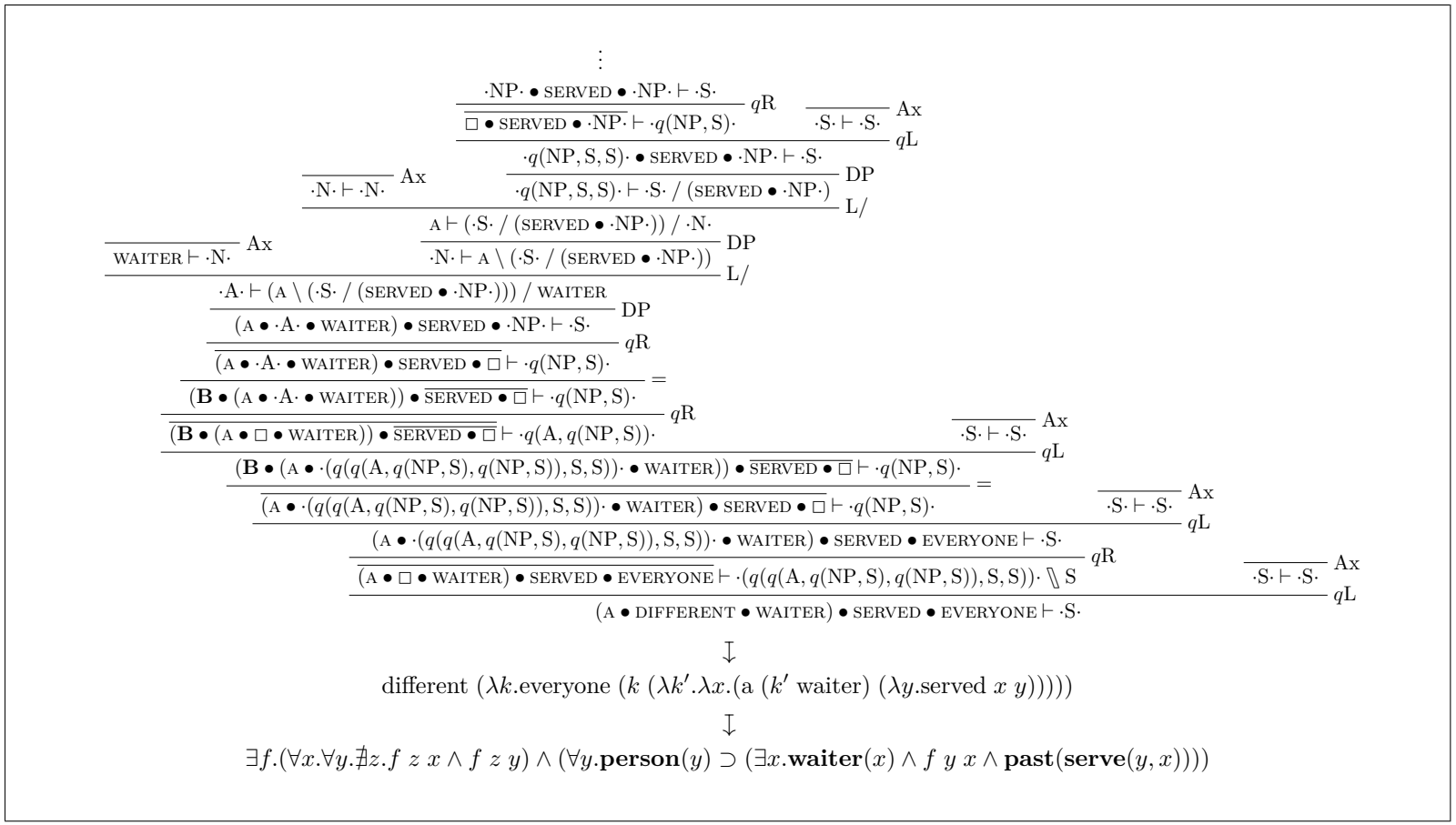


Figure 15: An example of parasitic scope.

4.3.4 Islands and diamonds

In section 4.2.3 we briefly discussed scope islands. Our goal for this section is to develop an extension to **IBC** for NLQ, which can be used to give 'to say' a type which blocks the wide-scope interpretation for the universal quantifier—creating an island from which it cannot escape.

The solution to this problem is rather simple: we have previously discussed that quantifiers are to be restricted to moving upwards and downwards past *solid products*. So what we should do is simply insert a structural connective which is not a solid product. From the literature on display calculus we can easily take a description of what a unary residuated connective should look like, and add this to the calculus. In figure 16 we describe such an extension, adding a dual pair of unary residuated connectives, $\Diamond$ and $\Box$.[25]

$$\begin{array}{rll} \text{Type} & A, B \coloneqq \ldots \mid \Diamond A \mid \Box A & \text{Pol}(\Diamond A) \mapsto + \\ \text{Structure}^{+} & \Gamma \coloneqq \ldots \mid \langle \Gamma \rangle & \text{Pol}(\Box B) \mapsto - \\ \text{Structure}^{-} & \Delta \coloneqq \ldots \mid [\Delta] & \end{array}$$

$$\frac{\langle \cdot A \cdot \rangle \vdash \Delta}{\cdot \Diamond A \cdot \vdash \Delta}\,\mathrm{L}\Diamond \qquad \frac{\Gamma \vdash \boxed{B}}{\langle \Gamma \rangle \vdash \boxed{\Diamond B}}\,\mathrm{R}\Diamond$$

$$\frac{\boxed{A} \vdash \Delta}{\boxed{\Box A} \vdash [\Delta]}\,\mathrm{L}\Box \qquad \frac{\Gamma \vdash [\cdot B \cdot]}{\Gamma \vdash \cdot \Box B \cdot}\,\mathrm{R}\Box$$

$$\frac{\Gamma \vdash [\Delta]}{\langle \Gamma \rangle \vdash \Delta}\,\mathrm{Res}\Box\Diamond$$

$$\Diamond A^{*} \mapsto A^{*} \quad \langle \Gamma \rangle^{*} \mapsto \Gamma^{*} \quad \langle \Gamma \rangle^{**} \mapsto \Gamma^{**}$$
$$\Box A^{*} \mapsto A^{*} \quad [\Delta]^{*} \mapsto \Delta^{*}$$

(all rules translate to the identity)

Figure 16: Extension of calculus in figure 12 which supports scope islands.

Using the newly defined connectives, we can assign 'said' the type $(\mathrm{NP} \setminus \mathrm{S}) / \Diamond \mathrm{S}$. Instead of taking a sentence-argument from the right, 'said' now takes a *closed-off* sentence—a scope island. Have a look at the derivation for "Mary said everyone left" given below:

[25] Note that these connectives have nothing to do with the unary connectives from modal logic, which are unfortunately also often called $\Diamond$ and $\Box$.

$$\dfrac{\dfrac{\dfrac{\vdots}{\textsc{everyone} \bullet \textsc{left} \vdash \cdot \mathrm{S} \cdot}}{\langle \textsc{everyone} \bullet \textsc{left} \rangle \vdash \cdot \Diamond \mathrm{S} \cdot}\,\mathrm{L}\Diamond \qquad \dfrac{\vdots}{\cdot \mathrm{NP} \setminus \mathrm{S} \cdot \vdash \textsc{mary} \setminus \cdot \mathrm{S} \cdot}}{\textsc{said} \vdash (\textsc{mary} \setminus \cdot \mathrm{S} \cdot) \,/\, \langle \textsc{everyone} \bullet \textsc{left} \rangle}\,\mathrm{L}/$$
$$\dfrac{\textsc{said} \vdash (\textsc{mary} \setminus \cdot \mathrm{S} \cdot) \,/\, \langle \textsc{everyone} \bullet \textsc{left} \rangle}{\textsc{mary} \bullet \textsc{said} \bullet \langle \textsc{everyone} \bullet \textsc{left} \rangle \vdash \cdot \mathrm{S} \cdot}\,\mathrm{DP}$$

$$\downarrow$$

say (everyone left) mary

$$\downarrow$$

$$\mathbf{say}(\mathbf{mary}, \forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{leave}(x)))$$

Note that in the bottom-most sequent, 'everyone' is nested under the structural scope island, and therefore cannot take scope at the top-level. Later on, when the L$\Diamond$-rule is applied, the scope island is deconstructed. However, it is already too late for 'everyone' to take sentence-wide scope: in feeding the scope island as an argument to 'said', one opens up the opportunity to deconstruct it, but at the same time isolates it from its surrounding context.

4.3.5 Strong versus weak quantifiers

There is one phenomenon that we, so far, have not addressed: the distinction between strong and weak quantifiers. Evidence for this distinction was discussed in section 4.2.3, but let us briefly examine the data again. The problem is that there is a difference in behaviour with respect to scope islands between two categories of quantifiers: strong quantifier (which include indefinites) and weak quantifiers (which includes universals). For instance, in "Mary said everyone left", we expect only the local-scope reading:

$$\mathbf{past}(\mathbf{say}(\mathbf{mary}, \forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{leave}(x))))$$

However, with "Mary said someone left", we expect both the local-scope and the wide-scope readings—the difference being that in the second reading, we can infer the existence of "someone":

$$\exists x.\mathbf{person}(x) \wedge \mathbf{past}(\mathbf{say}(\mathbf{mary}, \mathbf{past}(\mathbf{leave}(x))))$$
$$\mathbf{past}(\mathbf{say}(\mathbf{mary}, \exists x.\mathbf{person}(x) \wedge \mathbf{past}(\mathbf{leave}(x))))$$

More interestingly, with "Everyone said someone left", we expect to *also* get scope ambiguity due to the two quantifiers—the difference there being that everyone may have said "someone left", but they all may have been referring to different people:

$$\exists y.\mathbf{person}(y) \wedge \forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{say}(x, \mathbf{past}(\mathbf{leave}(y))))$$
$$\forall x.\mathbf{person}(x) \supset \exists y.\mathbf{person}(y) \wedge \mathbf{past}(\mathbf{say}(x, \mathbf{past}(\mathbf{leave}(y))))$$
$$\forall x.\mathbf{person}(x) \supset \mathbf{past}(\mathbf{say}(x, \exists y.\mathbf{person}(y) \wedge \mathbf{past}(\mathbf{leave}(y))))$$

There are two simple ways of getting NLQ to derive exactly these readings. The first is to extend the syntactic mechanisms for movement with a new combinator, analogous to **B** and **C**, which allows strong quantifiers to *syntactically* move past *weak* scope islands. The second solution would be to extend the CPS-semantics given by Moortgat and Moot (2011) to cover NLQ, and allow indefinites to take top-level scope *semantically*. We will discuss both solutions below.

Escaping islands with $\mathbf{I}^*$ Much in the style with Finger (1998) and Barker and Shan's (2014) encoding of linear lambda calculus with **I**, **B** and **C**, we can add a new combinator to encode scoping out of scope islands. For this, we will use the $\mathbf{I}^*$ combinator, which is the identity *over functions*. First, we do not want to allow *just any* quantifier to scope out of a delimiter. Therefore, we will split the existing quantifier modality into two separate modalities: weak quantifiers ($\{\backslash\!\!\backslash^w, \circ^w, /\!\!/^w\}$) and strong quantifiers ($\{\backslash\!\!\backslash^s, \circ^s, /\!\!/^s\}$), with copies of our existing rules for these connectives for both. We then add the new combinator $\mathbf{I}^*$ to our structure syntax, and add the following structural rule:

$$\frac{\langle \Gamma_1 \circ^s \Gamma_2 \rangle \vdash \Delta}{(\mathbf{I}^* \bullet \langle \Gamma_1 \rangle) \circ^s \Gamma_2 \vdash \Delta}\,\mathbf{I}^*$$

This rule encodes exactly the reduction behaviour of the $\mathbf{I}^*$-combinator:

$$\mathbf{I}^*\ xy \equiv xy$$

Using this approach, we can easily obtain the desired interpretations for "Everyone said someone left", since "someone"—a strong quantifier—can now scope out of the scope island put up by "said". As an added benefit, the explicit distinction between strong and weak quantifiers allows parasitic scope-takers to target only weak quantifiers, preventing the system from assigning a sentence-internal reading to the following sentence (as NL_λ does):

(*) "The same waiter served someone"

If it turns out to be necessary to define some form of scope islands from which even *strong* quantifiers cannot escape, one can easily extend this approach by also splitting the diamond/box pair used for scope islands into strong and weak versions, and updating the $\mathbf{I}^*$-rule to allow strong quantifiers to scope out of weak islands only.

This solution, including the extension with strong and weak delimiters is implemented in both the Agda and the Haskell verification of NLQ.

Indefinites and CPS-semantics Szabolcsi (2000) writes that "[i]ndefinites acquire their existential scope in a manner that does not involve movement and is essentially syntactically unconstrained." In light of this, it seems unlikely that we will have to extend our syntactic solution to include islands from which indefinites cannot escape—and our syntactic solution to weak versus strong quantifiers seems cumbersome.

It seems that a *combined* solution is in order to solve the distinction between strong and weak quantifiers. Our **IBC**-postulates serve us just fine for quantifiers such as "everyone", which really do seem to take scope by syntactic movement. However, for indefinites, a *semantic* solution seems in order. It is here that we recall that Bastenhof (2010), Moortgat and Moot (2011) have defined beautiful CPS-semantics for focused NL. We can easily extend this translation to cover our new structural postulates—we simply translate constants as units, and our logical left-unit $\mathbf{Q}(A)$ as A, as before. If we now adopt Moortgat and Moot's (2011) polarisation, we get *three* derivations for "Everyone said someone left":[26]

[26] If we want to assign "someone" the type $(\mathrm{NP} \,/\, \mathrm{N}) \otimes \mathrm{N}$, which has the expected quantificational effect, then we need to add products to NLQ.

1. We start by collapsing "someone" (which has it take top-level scope), continue by having "everyone" take scope (using our encoding of quantifier movement), and only then collapse the main clause "_ said _";

2. We start by having "everyone" take scope, then collapse "someone" (which has it take top-level scope), and once again by collapsing the main clause; or

3. We start by having "everyone" take scope, and then collapse *the main clause*, thereby forcing "someone" to take scope in the embedded clause instead.

This solution is implemented in the Agda verification of NLQ, and the final section of appendix A discusses the extended CPS-translation in more detail.

4.4 Extraction and gapped clauses

In the previous sections, we have discussed quantifier movement, which is characterised by upwards movement, and insertion of a trace. In this section, we will focus on simpler kinds of movement: infixation and extraction. These are types of movement where a constituent *only* moves down or *only* moves up.

In linguistic terms, 'extraction' means that in the syntax tree, a constituent is moved up from the position where it is interpreted. For instance,

(1) a. "character who meets his author"

b. "character whom Alice irritated"

(2) a. "book [the author of which] feared the ocean"

b. "book which Vonnegut dedicated to O'Hare"

In (1a), the "character is interpreted in the subject-position of the relative clause. For this example, it suffices to give 'which' the following definition:

$$\text{who} : ((\text{N} \setminus \text{N}) \,/\, (\text{NP} \setminus \text{S}))^*$$
$$\text{who} = \lambda f.\lambda g.\lambda x.f(x) \wedge g(x)$$

We used a similar solution in figure 13 to analyse (2a). In the presence of associativity, we could give a similar definition for 'whom' in (1b), moving the NP to the right branch of the relative clause, and counting on associativity to push it inwards:

$$\text{whom} : ((\text{N} \setminus \text{N}) \,/\, (\text{S} \,/\, \text{NP}))^*$$
$$\text{whom} = \lambda f.\lambda g.\lambda x.f(x) \wedge g(x)$$

However, we do not want to add unrestricted associativity. The phenomenon becomes even more complicated when you look at sentences such as (2b) where the "book" is interpreted deeply nested in the relative clause.

Moortgat (1999) describes a logical extension which handles all these cases of extraction—we present it in figure 17. The extension combines a fresh modality with a license—Moortgat (1999) adds a new diamond/box pair ($\diamond\downarrow$, $\square\downarrow$), and adds structural postulates which allow extraction to take place *only* in the

presence of the $\diamond\downarrow$. He then derives two new connectives, $\downarrow$ and $\upharpoonleft$, which we can use to give a definition for 'which' that can analyse (2b):

$$\text{which} : ((\text{N} \setminus \text{N}) \,/\, (\text{S} \downarrow \text{NP}))^*$$
$$\text{which} = \lambda f.\lambda g.\lambda x.f(x) \wedge g(x)$$

$$\begin{array}{rll} \text{Type} & A, B \coloneqq \ldots \mid \diamond\downarrow A \mid \square\downarrow A & \\ \text{Structure}^+ & \Gamma \coloneqq \ldots \mid \diamond\downarrow \Gamma & A \downarrow B \;\coloneqq\; (\diamond\downarrow \square\downarrow A) \setminus B \\ \text{Structure}^- & \Delta \coloneqq \ldots \mid \square\downarrow \Delta & B \upharpoonleft A \;\coloneqq\; B \,/\, (\diamond\downarrow \square\downarrow A) \end{array}$$

(copy of rules for $\{\diamond, \square\}$ from figure 16 for $\{\diamond\downarrow, \square\downarrow\}$)

$$\frac{\Gamma_1 \bullet (\Gamma_2 \bullet \diamond\downarrow \Gamma_3) \vdash \Delta}{(\Gamma_1 \bullet \Gamma_2) \bullet \diamond\downarrow \Gamma_3 \vdash \Delta}\,\text{RR}\diamond\downarrow \qquad \frac{(\Gamma_1 \bullet \diamond\downarrow \Gamma_3) \bullet \Gamma_2 \vdash \Delta}{(\Gamma_1 \bullet \Gamma_2) \bullet \diamond\downarrow \Gamma_3 \vdash \Delta}\,\text{LR}\diamond\downarrow$$

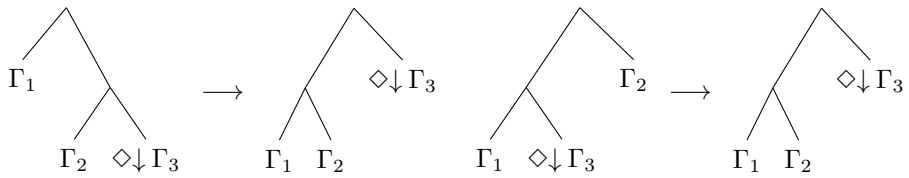
$$\frac{(\diamond\downarrow \Gamma_3 \bullet \Gamma_2) \bullet \Gamma_1 \vdash \Delta}{\diamond\downarrow \Gamma_3 \bullet (\Gamma_2 \bullet \Gamma_1) \vdash \Delta}\,\text{LL}\diamond\downarrow \qquad \frac{\Gamma_2 \bullet (\diamond\downarrow \Gamma_3 \bullet \Gamma_1) \vdash \Delta}{\diamond\downarrow \Gamma_3 \bullet (\Gamma_2 \bullet \Gamma_1) \vdash \Delta}\,\text{RL}\diamond\downarrow$$

(copy of translations for $\{\diamond, \square\}$ from figure 16 for $\{\diamond\downarrow, \square\downarrow\}$)

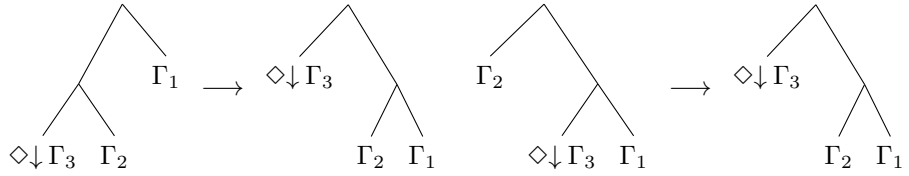
(RR$\diamond\downarrow$, LR$\diamond\downarrow$, LL$\diamond\downarrow$ and RL$\diamond\downarrow$ translate to various combinations of associativity and commutativity)

Figure 17: Extension of calculus in figure 7 which supports extraction.

It should be mentioned that the structural postulates for extraction differentiate between left-branch and right-branch extraction. RR$\diamond\downarrow$ and LR$\diamond\downarrow$ encode the right-branch movements:



On the other hand, LL$\diamond\downarrow$ and RL$\diamond\downarrow$ encode left-branch movements:



Moortgat (1999, sec. 1.2.1 and 1.2.2) further motivates the distinction between right- and left-branch extraction with more examples from English and Dutch.

One interesting consequence of this distinction for English is that we can make the distinction between 'who' and 'whom' on the type-level, without resorting to using case-marking. We assign 'which' an ambiguous type, as it can capture both subject and object-gaps. If we wish to cover (2a) *and* (2b), we can give 'which' the following definition:

$$\begin{aligned}&\text{which} : q(\text{NP}, \text{NP}, ((\text{N} \setminus \text{N}) \,/\, (\text{NP} \setminus \text{S})) \,\&\, ((\text{N} \setminus \text{N}) \,/\, (\text{S} \downharpoonleft \text{NP})))^*\\&\text{which} = \lambda f.(\lambda g.\lambda h.\lambda x.g(f(x)) \wedge h(x), \lambda g.\lambda h.\lambda x.g(f(x)) \wedge h(x))\end{aligned}$$

Or, if we adopt the additive disjunction ($\oplus$), as Lambek (1961), Morrill and Valentín (2015) suggest, we can give the following definition:

$$\begin{aligned}&\text{which} : q(\text{NP}, \text{NP}, ((\text{N} \setminus \text{N}) \,/\, ((\text{NP} \setminus \text{S}) \oplus (\text{S} \downharpoonleft \text{NP}))))^*\\&\text{which } f\ (\text{inj}_1\ g)\ h\ x = g(f(x)) \wedge h(x)\\&\text{which } f\ (\text{inj}_2\ g)\ h\ x = g(f(x)) \wedge h(x)\end{aligned}$$

Barker and Shan (2014, ch. 17.10) adopt two new inference rules in order to deal with gaps. However, these rules have a bit of an ad-hoc feel about them. They eliminate a logical connective, but also perform a structural function (movement) and they add an additional, unrelated meaning to the quantificational slash ($\backslash\!\backslash$):

$$\frac{\Sigma[A \bullet B] \vdash C}{\Sigma[A] \vdash B \mathbin{\backslash\!\backslash} C}\,\mathrm{R}_{\mathrm{rgap}} \qquad \frac{\Sigma[B \bullet A] \vdash C}{\Sigma[A] \vdash B \mathbin{\backslash\!\backslash} C}\,\mathrm{R}_{\mathrm{lgap}}$$

These rules are very similar to the rules we can derive for our extraction arrows:

$$\frac{\Sigma[\Gamma \bullet \cdot B \cdot] \vdash \cdot C \cdot}{\Sigma[\Gamma] \vdash \cdot C \downharpoonleft B \cdot}\,\mathrm{R}{\downharpoonleft} \qquad \frac{\Sigma[\cdot B \cdot \bullet \Gamma] \vdash \cdot C \cdot}{\Sigma[\Gamma] \vdash \cdot B \downharpoonright C \cdot}\,\mathrm{R}{\downharpoonright}$$

However, there are two important distinctions: (1) R$\downharpoonright$ and R$\downharpoonleft$ are *not* axiomatic rules, but they are derived from existing, *logical* connectives in display NL with extraction; and (2) R$\downharpoonright$ and R$\downharpoonleft$ distinguish between moving to the left- and right-hand side of the sub-structure. The first of these is an advantage, because it allows us to encode Barker and Shan's (2014) approach in our display calculus, in a more general manner, and without having to resort to proof search with contexts. The second one *may* be an advantage—there are be some situations where you require the distinction, and some where you do not. However, if you do not require the distinction, then you can always solve this using extensions for ambiguity (e.g. &) as we did for 'which'.

5 Related and Future work

Integrate Focusing and Display Logic

In section 2.4 we mentioned that, at the moment, there is no work which integrates focusing and display logic. Although we have a proof of cut-elimination for display NL and LG, due to Bastenhof (2011), we have not extended this proof to fully cover the system presented in this thesis. As mentioned in section 2.4, the reason for this is that in doing so, we would undo the advantage of using display calculus: if we have to maintain the proof of cut-elimination ourselves, then why use display calculus?

However, for the small number of natural language examples that were presented in this thesis, and those that were analysed in the Haskell implementation, focusing does exactly as advertised: it greatly reduces spurious ambiguity—eliminating it in many cases—while retaining all useful ambiguity. Therefore, it would be interesting to see the focusing integrated with display calculus, so that we would have a principled approach to developing focused display calculi.

An interesting approach to this is offered by Nigam et al. (2015), who present a procedure for generating focused systems and proofs of completeness form unfocused systems in an automated fashion.

Deep Inference Sequent Calculus

In section 2.2, we discussed that in order to use backward-chaining proof search, we have to ensure that our structural rules have the sub-structure property, or at very least only cause predictable loops. Dawson et al. (2014) demonstrate a methodology for constructing a deep inference sequent calculus from a display calculus. Deep inference calculi have the advantage that they naturally have the sub-structure property, which means that they are suitable for naive backward-chaining search. It would be interesting to employ this methodology, and construct a deep inference sequent calculus for the system constructed in this thesis.

Forward-Chaining Proof Search

In section 1.1 it was mentioned that most research focuses on implementing what I call the 'semantic function' (i.e. interpreting), and that we use backward-chaining proof search mostly because it is a pleasant tool for research purposes: it allows us to look to the huge body of work on generative grammar to inform our choices in parse trees, and focus our efforts on associating the right meanings to these known structures. However, in order to be feasible in a practical system, one must also implement what I call the 'syntactic function' (i.e. parsing).

The naive way to implement such a parsing algorithm, is to simply enumerate all the possible structures for a given sentence, and try them all. In practical parsing, however, this is not an option. The number of binary syntax trees for a sentence of n words is equal to the nth Catalan number.

A realistic way to implement parsing is by moving away from our backward-chaining proof search, and using forward-chaining proof search. For a naive implementation of this, we create a bag of axioms—one for each word—and construct all possible proofs that we can construct with only this set of axioms. Then we filter on those which are both pronounceable and maintain the correct word-order. Ideally, however, we would have an efficient implementation, for instance based on the technique of magic sets as developed by Bancilhon et al. (1986).

In section 4.3.4 we proposed to analyse scope islands with the unary diamond ($\Diamond$), which, in certain situations, enforces the presence of a structural diamond ($\langle \ldots \rangle$). It may be troubling to some of the readers that in order to deal with scope islands, we require that a structural connective is present in our input, in the endsequent. If we use forward-chaining proof search, however, this is much less problematic than it seems. When using forward-chaining search to look for

all proofs of e.g. "Mary said everyone left", the logical diamond in the type of 'said' will naturally ensure that the structural diamond is put into place, since they will be introduced symmetrically.

Integrate with Effectful Semantics

In sections 3 and 4 we discussed extensions of the syntactic calculus. However, $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$, too, has been extended and revisited many times. Many of its extensions were created to deal with complex semantic phenomena, such as intensionality (Ben-Avi and Winter, 2009), expressives (Potts, 2003; McCready, 2010; Gutzmann, 2011), and dynamic semantics (Groenendijk et al., 1995).

In 2002, Shan proposed an interesting paradigm to unify these extensions: by implementing them using techniques for effectful functional programming in $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$.

Shan proposed to analyse a wide range of linguistic phenomena using monads. He defines several monads which deal with interrogatives, focus, intensionality, binding, and quantification. Bumford (2013), Charlow (2014) and Barker and Shan (2014) continued this line of research, defining monads to deal with a large range of linguistic phenomena.

Formally, a monad is (1) a type-level constructor, $\mathbb{M}$, mapping each type A to a corresponding type $\mathbb{M}A$; and (2) a pair of functions, η and $\star$ (pronounced "unit" and "bind"), with the following types[27]:

$$\eta : A \to \mathbb{M}A \qquad \star : (A \to \mathbb{M}B) \to \mathbb{M}A \to \mathbb{M}B$$

There are many ways to implement monadic semantics. The most conventional of these is to apply the monadic translation, as described by Moggi (1991):

$$\begin{array}{ll} (A \to B)^{\mathrm{M}} & = A^M \to \mathbb{M}B^M \\ A^{\mathrm{M}} & = A \end{array} \qquad \begin{array}{ll} \mathrm{lift}\ x & = \eta\ x \\ \mathrm{lift}(\lambda x.M) & = \eta(\lambda x.\mathrm{lift}\ M) \\ \mathrm{lift}(M\ N) & = (\lambda f.f \star (\mathrm{lift}\ N)) \star \mathrm{lift}\ M \end{array}$$

Another possibility is to modify our translation to semantic calculus to insert the monadic operators. If we choose to do this, we can use the information present in out syntactic calculus to guide our translation. For instance, we could simply choose to modify our translation on types to insert an '$\mathbb{M}$' over every atomic type:

$$\mathrm{S}^* = \mathbb{M}\mathbf{t} \quad \mathrm{N}^* = \mathbb{M}(\mathbf{e} \to \mathbf{t}) \quad \mathrm{NP}^* = \mathbb{M}\mathbf{e}$$

Whichever choice we make, the important point is that the insertion of the monad constructor $\mathbb{M}$ in our types gives us the possibility to implement any sort of "plumbing" we need in our lexical entries, as long as it forms a monad.

As an example, we can use monads to analyse expressive content. This is content that is present in the sentence meaning, but does not directly affect the truth-conditional meaning. It is information present on a sort-of side channel. For instance, in "I walked the damn dog," the word 'damn' does not seem to

[27] In addition, these functions have to obey the monad laws: left identity ($M \star \eta N \equiv M\ N$); right identity ($\eta \star M \equiv M$); and associativity ($M \star (\lambda x.N\ x \star O) \equiv (M \star \lambda x.N\ x) \star O$).

contribute to the truth-conditional meaning, as the utterance would still be considered truthful if the dog is well-liked.

We can implement this using a variant of the writer monad: we represent values of the type $\mathbb{M}A$ as a tuple of truth-conditional (or "at-issue") content, and expressive content:

$$\begin{aligned}
\mathbb{M}A &= A \times \mathbf{t} \\
\eta\ M &= (M, \text{true}) \\
M \star N &= \text{case } M \text{ of } (x, y) \to (\text{case } N\ x \text{ of } (z, w) \to (z, y \wedge w)) \\
\text{tell}(M) &= ((), M)
\end{aligned}$$

Using this monad, we can define a small lexicon. We lift our regular entries into monadic entries:

$$\begin{aligned}
\text{john} &= \eta\ \mathbf{john} \\
\text{walks} &= \lambda y\ x.(\lambda x'.(\lambda y'.\eta(\mathbf{walk}(x, y))) \star y) \star x \\
\text{the} &= \lambda f.(\lambda f'.\iota(f')) \star f \\
\text{dog} &= \eta\ \mathbf{dog}
\end{aligned}$$

We treat 'damn' as an identity function in its at-issue content—it binds x', then returns it. However, we also define 'damn' as expressing some sort of displeasure, represented as the proposition **damn** in its expressive content:

$$\text{damn} = \lambda f.(\lambda f'.(\lambda().\eta\ f') \star \text{tell}(\mathbf{damn}(f'))) \star f$$

The entire utterance "I walked the damn dog" then reduces as follows:

$$(\text{walks } (\text{the } (\text{damn dog})) \text{ john}) \mapsto (\mathbf{walk}(\mathbf{john}, \iota(\mathbf{dog})), \mathbf{damn}(\mathbf{dog}))$$

The above analysis is rather coarse, as it does not capture any displeasure towards the *specific* dog. We *can* get a more precise meaning, but doing so complicates the example too much.

Monads have one big problem: modularity. There is no general procedure which can compose two arbitrary monads $\mathbb{M}_1$ and $\mathbb{M}_2$ into a new monad $\mathbb{M}_3 = \mathbb{M}_1 \circ \mathbb{M}_2$. This means that it is not trivial to separate different types of effects—*all* side-effects have to be implemented in one single, monolithic monad.

Shan (2002) mentions monad morphisms or transformers as a solution to mitigate—but not solve—the problem. Monad transformers were introduced by Liang et al. (1995). In short, they are functions $\mathbb{T}$ from monads to monads. Transformers can be chained together, to create combined monads consisting of "layers" of elementary monads. Because different monads combine in different ways, the programmer has to manually define these transformers, and has to to specify how effectful operations 'lift' through each monad transformer. One problem with monad transformers is that the order of the "layers" is determined statically, and cannot easily be changed in various parts of the program. In addition, every effectful operation has to be lifted into this layercake of side-effects. This means that that every effectful function, or lexical entry, has access to *all* side-effects, and every effectful function has to be altered if a new layer

is added. It is clear that monad transformers offer a sub-par solution to the problem.[28]

Cartwright and Felleisen (1994), Kiselyov et al. (2013) and Kiselyov and Ishii (2015) offer a solution to the problem of modularity, in the form of *extensible effects*. An in-depth discussion of extensible effects is beyond the scope of this thesis, so we will simply give an outline of the interface presented by the 2015 implementation of extensible effects.

In short, this implementation a type constructor $\mathbb{E}_X$ which is indexed by a "set" of effects. This constructor forms a monad for arbitrary sets X, so we can keep using the style of lexical definitions we saw above. What sets extensible effects apart from monad transformers is that with extensible effects, instead of defining a transformer and a lifting operator, the programmer defines an 'effect' in isolation from every other effect. An effect is defined by three things: 1) a type constructor, which links the type of effectful values to the type of the effect; 2) primitive effectful functions, such as 'tell'; 3) a handler, which removes the effect from the set of effects, optionally consuming or producing additional input or output. In the case of expressive content, the effect is defined as follows:[29]

$$\mathrm{Exp}\ \top = \mathbf{t}$$

Once we have this definition, we can recover the 'tell' function using one of the primitives offered by extensible effects—the 'send' function. In fact, 'tell' is exactly the 'send' function, with 'F' instantiated to the expressive effect:

$$\mathrm{send} : FA \to \mathbb{E}_{\{F\}\cup X} A \quad \textit{specialises to} \quad \mathrm{tell} : \mathbf{t} \to \mathbb{E}_{\{Exp\}\cup X} \top$$

What we have gained at this point is the parameter X—the 'tell' function is now polymorphic in the set of effects. This means that a word with only expressive content—such as 'damn'—only has access to the effects associated with expressive content, and not—as was the case with monad transformers—to the entire stack of effects. Last, we have to define a handler for the effect. This means defining a function which takes a value which includes the effect, and returns a value without it. For expressives, our handler will have the following type:

$$\mathrm{run}_{\mathrm{Exp}} : \mathbb{E}_{\{Exp\}\cup X} A \to \mathbb{E}_X (A \times \mathbf{t})$$

This handler will remove the expressive effect, and tuples the expressive content with the at-issue content. In general, handlers allow us to remove effects from the set of effects step-by-step until we once again end up with an effect-free value.

Interestingly, we named two limitations of the CPS-translation approach to quantification: the inability to change the answer type, and the inability to encode delimiters. These are the hallmark of *delimited* or *composable* continuations (Danvy and Filinski, 1990). However, while delimited continuations seem extremely promising, they are still not entirely without problems. They still

[28] The Haskell community is split over whether or not monad transformers are useful in practice, but many people—including the GHC developers—prefer "rolling" their own monolithic monad, which includes all required effects, over using monad transformers (see `http://stackoverflow.com/a/2760709`).

[29] The usage of $\top$ in the definition of Exp means that Exp is *only* defined for the $\top$ type—this, in turn, forces the output type of 'tell' to be $\top$.

suffer from the problem of ambiguity, as described above for CPS- and monadic translations. Additionally, they do not form a monad. Instead, they form something known as an indexed monad, which has two additional type-level parameters, meaning η and $\star$ get the following types:

$$\eta : A \to \mathbb{M}\ i\ i\ A \qquad \star : (A \to \mathbb{M}\ j\ k\ B) \to \mathbb{M}\ i\ j\ A \to \mathbb{M}\ i\ k\ B$$

This makes sense: since we now allow the answer type of the continuation to change, we need to add indices to keep track of the input and output answer type. However, the downside of this is that since we need these additional parameters, we cannot simply CPS-translate to delimited continuations—if we use a delimited continuation indexed monad in our semantics, this will have to be reflected in our syntactic calculus.

References

Andreoli, J. (1992). Logic programming with focusing proofs in linear logic. *J. Log. Comput.*, 2(3):297–347.

Bancilhon, F., Maier, D., Sagiv, Y., and Ullman, J. D. (1986). Magic sets and other strange ways to implement logic programs. In Silberschatz, A., editor, *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, March 24-26, 1986, Cambridge, Massachusetts, USA*, pages 1–15. ACM.

Barker, C. (2002). Continuations and the nature of quantification. *Natural Language Semantics*, 10(3):211–242.

Barker, C. (2004). Continuations in natural language. *CW*, 4:1–11.

Barker, C. (2007). Parasitic scope. *Linguistics and Philosophy*, 30(4):407–444.

Barker, C. and Shan, C. (2014). *Continuations and Natural Language*, volume 53 of *Oxford Studies in Theoretical Linguistics*. Oxford University Press.

Bastenhof, A. (2010). Polarized montagovian semantics for the lambek-grishin calculus. In de Groote, P. and Nederhof, M., editors, *Formal Grammar - 15th and 16th International Conferences, FG 2010, Copenhagen, Denmark, August 2010, FG 2011, Ljubljana, Slovenia, August 2011, Revised Selected Papers*, volume 7395 of *Lecture Notes in Computer Science*, pages 1–16. Springer.

Bastenhof, A. (2011). Polarized classical non-associative Lambek calculus and formal semantics. In Pogodalla, S. and Prost, J., editors, *Logical Aspects of Computational Linguistics - 6th International Conference, LACL 2011, Montpellier, France, June 29 - July 1, 2011. Proceedings*, volume 6736 of *Lecture Notes in Computer Science*, pages 33–48. Springer.

Bastenhof, A. et al. (2013). Categorial symmetry. `http://dspace.library.uu.nl/handle/1874/273870`.

Ben-Avi, G. and Winter, Y. (2009). Scope dominance with generalized quantifiers. In Grumberg, O., Kaminski, M., Katz, S., and Wintner, S., editors, *Languages: From Formal to Natural*, volume 5533 of *Lecture Notes in Computer Science*, pages 36–44. Springer Berlin Heidelberg.

Bernardi, R. and Moortgat, M. (2010). Continuation semantics for the lambek-grishin calculus. *Inf. Comput.*, 208(5):397–416.

Bumford, D. (2013). Universal quantification as iterated conjunction. In *Logic, Language and Meaning. Proceedings of the 19th Amsterdam Colloquium*, pages 67–74.

Capelletti, M. (2007). Parsing with structure-preserving categorial grammars. `http://dspace.library.uu.nl/handle/1874/21930`.

Cartwright, R. and Felleisen, M. (1994). Extensible denotational language specifications. In Hagiya, M. and Mitchell, J. C., editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 244–272. Springer.

Cervesato, I. and Schürmann, C., editors (2015). *Proceedings First International Workshop on Focusing, WoF'15 2015, Suva, Fiji, 23rd November 2015*, volume 197 of *EPTCS*.

Charlow, S. (2014). On the semantics of exceptional scope.

Curien, P. and Herbelin, H. (2000). The duality of computation. In Odersky, M. and Wadler, P., editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, pages 233–243. ACM.

Danvy, O. and Filinski, A. (1990). Abstracting control. In *LISP and Functional Programming*, pages 151–160.

Dawson, J. E., Clouston, R., Goré, R., and Tiu, A. (2014). From display calculi to deep nested sequent calculi: Formalised for full intuitionistic linear logic. In Diaz, J., Lanese, I., and Sangiorgi, D., editors, *Theoretical Computer Science - 8th IFIP TC 1/WG 2.2 International Conference, TCS 2014, Rome, Italy, September 1-3, 2014. Proceedings*, volume 8705 of *Lecture Notes in Computer Science*, pages 250–264. Springer.

Eisenberg, R. A. and Weirich, S. (2012). Dependently typed programming with singletons. pages 117–130.

Finger, M. (1998). Computational solutions for structural constraints. In Moortgat, M., editor, *Logical Aspects of Computational Linguistics, Third International Conference, LACL'98, Grenoble, France, December 14-16, 1998, Selected Papers*, volume 2014 of *Lecture Notes in Computer Science*, pages 11–30. Springer.

Girard, J.-Y. (1987). Linear logic. *Theoretical Computer Science*, 50:1–102.

Girard, J.-Y. (1991). A new constructive logic: classic logic. *Math. Struct. in Comp. Science*, 1(03):255.

Goré, R. (1998). Substructural logics on display. *Logic Journal of the IGPL*, 6(3):451–504.

Groenendijk, J., Stokhof, M., and Veltman, F. (1995). *Coreference and modality*. Institute for Logic, Language and Computation (ILLC), University of Amsterdam.

Gutzmann, D. (2011). Expressive modifiers and mixed expressives. *Empirical issues in syntax and semantics*, 8:123–141.

Hendriks, H. L. W. (1993). *Studied flexibility: Categories and types in syntax and semantics*. Institute for Logic, Language and Computation.

Hepple, M. (1990). Normal form theorem proving for the lambek calculus. In *13th International Conference on Computational Linguistics, COLING 1990, University of Helsinki, Finland, August 20-25, 1990*, pages 173–178.

Jr., N. D. B. (1982). Display logic. *J. Philosophical Logic*, 11(4):375–417.

Kanazawa, M. (1992). The lambek calculus enriched with additional connectives. *Journal of Logic, Language and Information*, 1(2):141–171.

Kiselyov, O. (2015). Compositional semantics of *same*, *different*, *total*. In *Proceedings for ESSLLI 2015 Workshop 'Empirical Advances in Categorial Grammar'*, pages 71–81.

Kiselyov, O. and Ishii, H. (2015). Freer monads, more extensible effects. In Lippmeier, B., editor, *Proceedings of the 8th ACM SIGPLAN Symposium on Haskell, Haskell 2015, Vancouver, BC, Canada, September 3-4, 2015*, pages 94–105. ACM.

Kiselyov, O., Sabry, A., and Swords, C. (2013). Extensible effects: an alternative to monad transformers. pages 59–70.

Kiselyov, O. and Shan, C.-c. (2014). Continuation hierarchy and quantifier scope. In *Formal Approaches to Semantics and Pragmatics*, pages 105–134. Springer.

Lambek, J. (1958). The mathematics of sentence structure. *The American Mathematical Monthly*, 65(3):pp. 154–170.

Lambek, J. (1961). On the calculus of syntactic types. *Structure of language and its mathematical aspects*, 166:C178.

Laurent, O. (2004). A proof of the focalization property of linear logic. *Unpublished note, May*.

Liang, S., Hudak, P., and Jones, M. (1995). Monad transformers and modular interpreters. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 333–343, New York, NY, USA. ACM.

Marlow, S. (2012). Haskell 2010 language report.

McBride, C. T. (2014). How to keep your neighbours in order. In Jeuring, J. and Chakravarty, M. M. T., editors, *Proceedings of the 19th ACM SIGPLAN international conference on Functional programming, Gothenburg, Sweden, September 1-3, 2014*, pages 297–309. ACM.

McCready, E. S. (2010). Varieties of conventional implicature. *Semantics and Pragmatics*, 3:8–1.

Moggi, E. (1991). Notions of computation and monads. *Inf. Comput.*, 93(1):55–92.

Montague, R. (1973). The proper treatment of quantification in ordinary english. In *Approaches to Natural Language*, pages 221–242. Springer.

Moortgat, M. (1996). In situ binding: A modal analysis. In Dekker, P. and Stokhof, M., editors, *Proceedings of the Tenth Amsterdam Colloquium*, pages 539–549. Institute for Logic, Language and Computation (ILLC).

Moortgat, M. (1999). Constants of grammatical reasoning. In *Constraints and Resources in Natural Language Syntax and Semantics*, pages 195–219. Publications.

Moortgat, M. (2009). Symmetric categorial grammar. *J. Philosophical Logic*, 38(6):681–710.

Moortgat, M. and Moot, R. (2011). Proof nets for the Lambek-Grishin calculus. volume abs/1112.6384.

Moortgat, M. and Oehrle, R. T. (1999). Proof nets for the grammatical base logic. In Abrusci, M. and Casadio, C., editors, *Dynamic perspectives in logic and linguistics. Proceedings of the Fourth Roma Workshop*, pages 131–143, Roma. Bulzoni.

Moot, R. and Retoré, C. (2012). *The logic of categorial grammars: a deductive account of natural language syntax and semantics*, volume 6850. Springer.

Morrill, G. (1994). *Type logical grammar - categorial logic of signs*. Kluwer.

Morrill, G. and Valentín, O. (2015). Multiplicative-additive focusing for parsing as deduction. In Cervesato and Schürmann (2015), pages 29–54.

Nigam, V., Reis, G., and Lima, L. (2015). Towards the automated generation of focused proof systems. In Cervesato and Schürmann (2015), pages 1–6.

Norell, U. (2009). Dependently typed programming in agda. In Kennedy, A. and Ahmed, A., editors, *Proceedings of TLDI'09: 2009 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Savannah, GA, USA, January 24, 2009*, pages 1–2. ACM.

Parigot, M. (1992). $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In Voronkov, A., editor, *Logic Programming and Automated Reasoning,International Conference LPAR'92, St. Petersburg, Russia, July 15-20, 1992, Proceedings*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer.

Potts, C. (2003). Expressive content as conventional implicature. In *Proceedings-Nels*, volume 33, pages 303–322.

Shan, C. (2002). Monads for natural language semantics. *CoRR*, cs.CL/0205026.

Szabolcsi, A. (2000). The syntax of scope. In *The Handbook of Contemporary Syntactic Theory*, pages 607–633. Wiley-Blackwell.

A Formalisation of NLQ in Agda

In this appendix, we will discuss the Agda formalisation of *focused* NLQ. This section is written in literate Agda, and includes *all* of the code. The order of presentation is different from the order used in the thesis, due to constraints on the Agda language.

In the first part of this appendix, we will formalise the syntactic calculus, NLQ. Then, in the second part, we will implement a translation from proofs in NLQ to terms in Agda, giving us some form of semantics. But first, we will discuss our motivation in deciding to formalise our work.

A.1 Motivation

Why would we want to formalise type-logical grammars using proof assistants? One good reason is that it allows us to write formally verified proofs about the theoretical properties of our type-logical grammars. But not only that—it allows us to directly run our proofs as programs. For instance, we can directly run the translation from NLQ to Agda, presented in this paper, to investigate what kind of derivations result in what kind of semantics, *and* be confident in its correctness. In addition, we will be able to use any interactive theorem prover that our proof assistant of choice provides to experiment with and give proofs in our type-logical grammar.

Why, then, would we want to use Agda instead of a more established proof assistant such as, for instance, Coq? There are several good reasons, but we believe that the syntactic freedom offered by Agda is the most important. It is this freedom that allows us to write machine-checkable proofs, formatted in a way which is very close to the way one would otherwise typeset proofs, and which are highly readable compared to other machine-checked proofs. This is true to a lesser extend for the formalisation presented in this appendix, since we forgo a number of features that generally make Agda code more readable in order to stay as close as possible to the Haskell implementation in appendix B. However, we do feel that, the Agda verification of the theory presented here can be of great use in understanding thxe Haskell code.

The addition of these proofs means that we can be confident that the proofs *as they are published* are correct, and that they are necessarily complete—for though we can hide some of the less interesting definitions from the final paper, we cannot omit them from the source. In addition, some of the Agda proofs serve as explicit and fully formal versions of proofs that were merely hinted at in the thesis. Other Agda proofs serve as justification for Haskell functions which circumvent the type-system—as Agda's type system is vastly more powerful than that of Haskell.

Finally, because there is a correspondence between the published proofs and the code, it becomes very easy for the reader to start up a proof environment and inspect the proofs interactively in order to further their understanding of the presented work.

A.2 NLQ, Syntactic Calculus

For our formalisation of NLQ, we are going to abstract over the atomic types—a luxury offered by the Agda module system. The reason for this is that the set

of atomic types is more-or-less open—sometimes we find out that we have to add a new one—and can be treated in a uniform manner. Because atomic types must be assigned a polarity, we will start out by defining a notion of polarity and polarised values:

```
data Polarity : Set where
  + : Polarity
  − : Polarity

~_ : Polarity → Polarity
~ + = −
~ − = +

record Polarised (A : Set) : Set where
  field
    Pol : A → Polarity
open Polarised {{...}}
```

We can now open our Syntax module, abstracting over the type of atomic types and a notion of polarisation for this type:

```
module Syntax (Atom : Set) (PolarisedAtom : Polarised Atom) where
```

A.2.1 Types, Structures and Sequents

First thing to do is to define our types. We abstract a little here: instead of defining several copies of our rules for $\{\backslash, \bullet, /\}$ and $\{\Diamond, \Box\}$ for new connectives, as we did in the thesis, we define a datatype to represent the different kinds of connectives we will be using, and parameterise our connectives with a kind. We then recover the pretty versions of our connectives using pattern synonyms. The advantage of this approach is that we can later-on use e.g. the abstract right implication ImpR in the definitions of the inference rules, defining all the copies at the same time.

```
  data Strength : Set where
    Weak   : Strength
    Strong : Strength

  data Kind : Set where
    Solid  : Kind                 - solid {\, •, /}
    Quan   : Strength → Kind      - hollow {⧵, ∘, ⫽}
    Del    : Strength → Kind      - reset {◇, □}
    Ifx    : Kind                 - extraction {↑, ↾, ◇↑, □↑}
    Ext    : Kind                 - infixation {↓, ⇂, ◇↓, □↓}

  data Type : Set where
    El     : Atom → Type
    Dia    : Kind → Type → Type
    Box    : Kind → Type → Type
    UnitL  : Kind → Type → Type
```

```
ImpR : Kind → Type → Type → Type
ImpL : Kind → Type → Type → Type
```

```
pattern _/_   b a = ImpL Solid          b a
pattern _\_   a b = ImpR Solid          a b
pattern _⫽_   a b = ImpR (Quan Weak)    a b
pattern _⫻_   b a = ImpL (Quan Weak)    b a
pattern QW    a   = UnitL (Quan Weak)   a
pattern QS    a   = UnitL (Quan Strong ) a
pattern ◇_    a   = Dia  (Del Weak)  a
pattern ◇↑_   a   = Dia  Ifx          a
pattern ◇↓_   a   = Dia  Ext          a
pattern □_    a   = Box  (Del Weak)  a
pattern □↑_   a   = Box  Ifx          a
pattern □↓_   a   = Box  Ext          a

pattern _⇂_   a b = ◇↑ □↑ (a \ b)
pattern _↾_   b a = ◇↑ □↑ (b / a)
pattern _⇃_   a b = (◇↓ □↓ a) \ b
pattern _↿_   b a = b / (◇↓ □↓ a)
```

We use the same trick in defining structures, and merge Struct^+ and Struct^- together into a single datatype indexed by a polarity:

```
data Struct : Polarity → Set where
  ·_·     : Type → Struct p
  B       : Struct +
  C       : Struct +
  I*      : Struct +
  DIA     : Kind → Struct +  → Struct +
  UNIT    : Kind → Struct +
  PROD    : Kind → Struct +  → Struct +  → Struct +
  BOX     : Kind → Struct −  → Struct −
  IMPR    : Kind → Struct +  → Struct −  → Struct −
  IMPL    : Kind → Struct −  → Struct +  → Struct −
```

```
pattern _•_   x y = PROD Solid          x y
pattern _∘_   x y = PROD (Quan Weak)    x y
pattern _∘ˢ_  x y = PROD (Quan Strong)  x y
pattern ⟨_⟩   x   = DIA  (Del Weak)     x
pattern ⟨_⟩ˢ  x   = DIA  (Del Strong)   x
pattern ⧫↑_   x   = DIA  Ifx            x
pattern ⧫↓_   x   = DIA  Ext            x
pattern I         = UNIT (Quan Weak)
pattern ■↑_   x   = BOX  Ifx            x
pattern ■↓_   x   = BOX  Ext            x
```

Since there is no pretty way to write the box we used for focusing in Unicode,

we will have to go with an ugly way:

```
data Sequent : Set where
  _⊢_    : Struct +  → Struct −  → Sequent
  [_]⊢_  : Type      → Struct −  → Sequent
  _⊢[_]  : Struct +  → Type      → Sequent
```

And finally, we need to extend our concept of polarity to *types*:

```
instance
  PolarisedType : Polarised Type
  PolarisedType = record { Pol = Pol′ }
    where
      Pol′ : Type → Polarity
      Pol′ (El    a)       = Pol(a)
      Pol′ (Dia   _ _)     = +
      Pol′ (Box   _ _)     = −
      Pol′ (UnitL _ _)     = +
      Pol′ (ImpR  _ _ _)   = −
      Pol′ (ImpL  _ _ _)   = −
```

A.2.2 Inference Rules

Below we define the logic of NLQ as a single datatype, indexed by a sequent. As described in the section on focusing, the axioms and focusing/unfocusing rules take an extra argument—a piece of evidence for the polarity of the type a/b:

```
data NLQ_ : Sequent → Set where
  axElR  : Pol(b) ≡ +  → NLQ · El b · ⊢[ El b ]
  axElL  : Pol(a) ≡ −  → NLQ [ El a ]⊢ · El a ·
  unfR   : Pol(b) ≡ −  → NLQ x ⊢ · b ·  → NLQ x ⊢[ b ]
  unfL   : Pol(a) ≡ +  → NLQ · a · ⊢ y  → NLQ [ a ]⊢ y
  focR   : Pol(b) ≡ +  → NLQ x ⊢[ b ]   → NLQ x ⊢ · b ·
  focL   : Pol(a) ≡ −  → NLQ [ a ]⊢ y   → NLQ · a · ⊢ y

  impRL  : NLQ x ⊢[ a ]  → NLQ [ b ]⊢ y  → NLQ [ ImpR k a b ]⊢ IMPR k x y
  impRR  : NLQ x ⊢ IMPR k · a · · b ·    → NLQ x ⊢ · ImpR k a b ·
  impLL  : NLQ x ⊢[ a ]  → NLQ [ b ]⊢ y  → NLQ [ ImpL k b a ]⊢ IMPL k y x
  impLR  : NLQ x ⊢ IMPL k · b · · a ·    → NLQ x ⊢ · ImpL k b a ·
  resRP  : NLQ y ⊢ IMPR k x z  → NLQ PROD k x y ⊢ z
  resPR  : NLQ PROD k x y ⊢ z  → NLQ y ⊢ IMPR k x z
  resLP  : NLQ x ⊢ IMPL k z y  → NLQ PROD k x y ⊢ z
  resPL  : NLQ PROD k x y ⊢ z  → NLQ x ⊢ IMPL k z y

  diaL   : NLQ DIA k · a · ⊢ y  → NLQ · Dia k a · ⊢ y
  diaR   : NLQ x ⊢[ b ]         → NLQ DIA k x ⊢[ Dia k b ]
  boxL   : NLQ [ a ]⊢ y         → NLQ [ Box k a ]⊢ BOX k y
  boxR   : NLQ x ⊢ BOX k · b ·  → NLQ x ⊢ · Box k b ·
  resBD  : NLQ x ⊢ BOX k y      → NLQ DIA k x ⊢ y
  resDB  : NLQ DIA k x ⊢ y      → NLQ x ⊢ BOX k y
```

```
unitLL  : NLQ PROD k (UNIT k) · a · ⊢ y → NLQ · UnitL k a · ⊢ y
unitLR  : NLQ x ⊢[ b ] → NLQ PROD k (UNIT k) x ⊢[ UnitL k b ]
unitLI  : NLQ x ⊢ y → NLQ PROD k (UNIT k) x ⊢ y

dnB     : NLQ x • (PROD (Quan k) y z) ⊢ w
        → NLQ PROD (Quan k) ((B • x) • y) z ⊢ w
upB     : NLQ PROD (Quan k) ((B • x) • y) z ⊢ w
        → NLQ x • (PROD (Quan k) y z) ⊢ w
dnC     : NLQ (PROD (Quan k) x y) • z ⊢ w
        → NLQ PROD (Quan k) ((C • x) • z) y ⊢ w
upC     : NLQ PROD (Quan k) ((C • x) • z) y ⊢ w
        → NLQ (PROD (Quan k) x y) • z ⊢ w
upI*    : NLQ ((I* • ⟨ x ⟩) ∘ˢ y ⊢ w) → NLQ (⟨ x ∘ˢ y ⟩ ⊢ w)
dnI*    : NLQ (⟨ x ∘ˢ y ⟩ ⊢ w) → NLQ ((I* • ⟨ x ⟩) ∘ˢ y ⊢ w)

ifxRR   : NLQ ((x • y) • ⧫↑ z ⊢ w)  → NLQ (x • (y • ⧫↑ z) ⊢ w)
ifxLR   : NLQ ((x • y) • ⧫↑ z ⊢ w)  → NLQ ((x • ⧫↑ z) • y ⊢ w)
ifxLL   : NLQ (⧫↑ z • (y • x) ⊢ w)  → NLQ ((⧫↑ z • y) • x ⊢ w)
ifxRL   : NLQ (⧫↑ z • (y • x) ⊢ w)  → NLQ (y • (⧫↑ z • x) ⊢ w)

extRR   : NLQ (x • (y • ⧫↓ z) ⊢ w)  → NLQ ((x • y) • ⧫↓ z ⊢ w)
extLR   : NLQ ((x • ⧫↓ z) • y ⊢ w)  → NLQ ((x • y) • ⧫↓ z ⊢ w)
extLL   : NLQ ((⧫↓ z • y) • x ⊢ w)  → NLQ (⧫↓ z • (y • x) ⊢ w)
extRL   : NLQ (y • (⧫↓ z • x) ⊢ w)  → NLQ (⧫↓ z • (y • x) ⊢ w)
```

Using these axiomatic rules, we can define derived rules. For instance, we can define the following "residuation" rules, which convert left implication to right implication, and vice versa:

```
resRL : NLQ y ⊢ IMPR k x z → NLQ x ⊢ IMPL k z y
resRL f = resPL (resRP f)

resLR : NLQ x ⊢ IMPL k z y → NLQ y ⊢ IMPR k x z
resLR f = resPR (resLP f)
```

A.2.3 Contexts and Plugging functions

NLQ might not need contexts and plugging functions for its specification, but many meta-logical proofs nonetheless require this vocabulary. In preparation for the proof in the following section, We will therefore define a notion of contexts for NLQ. We start by defining contexts an class of "pluggable"' things:

```
record Pluggable (C I O : Set) : Set where
  field
    _[_] : C → I → O
open Pluggable {{...}}
```

Next, we define the first type of context: full structural contexts, i.e. structures with a single hole. For this, we simply replicate the structure of contexts, and

add the HOLE-constructor. Note that we replicate binary constructors twice—once with the hole to the left, and once with the hole to the right:

```
data StructCtxt (p : Polarity) : Polarity → Set where
  HOLE  : StructCtxt p p
  DIA1  : Kind → StructCtxt p +  → StructCtxt p +
  PROD1 : Kind → StructCtxt p +  → Struct    +  → StructCtxt p +
  PROD2 : Kind → Struct    +  → StructCtxt p +  → StructCtxt p +
  BOX1  : Kind → StructCtxt p −  → StructCtxt p −
  IMPR1 : Kind → StructCtxt p +  → Struct    −  → StructCtxt p −
  IMPR2 : Kind → Struct    +  → StructCtxt p −  → StructCtxt p −
  IMPL1 : Kind → StructCtxt p −  → Struct    +  → StructCtxt p −
  IMPL2 : Kind → Struct    −  → StructCtxt p +  → StructCtxt p −
```

Plugging is simply the process of taking a given structure, and inserting this in place of the hole:

```
instance
  Pluggable-Struct : Pluggable (StructCtxt p1 p2) (Struct p1) (Struct p2)
  Pluggable-Struct = record { _[_] = _[_]′ }
    where
      _[_]′ : StructCtxt p1 p2 → Struct p1 → Struct p2
      ( HOLE          ) [ z ]′ = z
      ( DIA1  k x     ) [ z ]′ = DIA  k   (x [ z ]′)
      ( PROD1 k x y   ) [ z ]′ = PROD k   (x [ z ]′) y
      ( PROD2 k x y   ) [ z ]′ = PROD k x (y [ z ]′)
      ( BOX1  k x     ) [ z ]′ = BOX  k   (x [ z ]′)
      ( IMPR1 k x y   ) [ z ]′ = IMPR k   (x [ z ]′) y
      ( IMPR2 k x y   ) [ z ]′ = IMPR k x (y [ z ]′)
      ( IMPL1 k x y   ) [ z ]′ = IMPL k   (x [ z ]′) y
      ( IMPL2 k x y   ) [ z ]′ = IMPL k x (y [ z ]′)
```

In accordance with our approach in the previous sections, we recover more specific (and prettier) context-constructors using pattern synonyms:

```
pattern _<•_   x y = PROD1 Solid          x y
pattern _<\_   x y = IMPR2 Solid          x y
pattern _</_   y x = IMPL1 Solid          y x
pattern _<∘_   x y = PROD1 (Quan Weak)    x y
pattern _<⫬_   x y = IMPR1 (Quan Weak)    x y
pattern _<⫽_   y x = IMPL1 (Quan Weak)    y x
pattern _•>_   x y = PROD2 Solid          x y
pattern _\>_   x y = IMPR2 Solid          x y
pattern _/>_   y x = IMPL1 Solid          y x
pattern _∘>_   x y = PROD2 (Quan Weak)    x y
pattern _⫬>_   x y = IMPR2 (Quan Weak)    x y
pattern _⫽>_   y x = IMPL2 (Quan Weak)    y x
pattern ♦>_    x   = DIA1  (Del Weak)     x
pattern ♦↓>_   x   = DIA1  Ifx            x
pattern ♦↑>_   x   = DIA1  Ext            x
```

```
pattern ■>_    x   = BOX1   (Del Weak)   x
pattern ■↓>_   x   = BOX1   Ifx          x
pattern ■↑>_   x   = BOX1   Ext          x
```

And we do the same for sequents:

```
data SequentCtxt (p : Polarity) : Set where
  _<⊢_ : StructCtxt p + → Struct     − → SequentCtxt p
  _⊢>_ : Struct     + → StructCtxt p − → SequentCtxt p

instance
  Pluggable-Sequent : Pluggable (SequentCtxt p) (Struct p) Sequent
  Pluggable-Sequent = record { _[_] = _[_]′ }
    where
      _[_]′ : SequentCtxt p → Struct p → Sequent
      (x <⊢ y) [ z ]′ = x [ z ] ⊢ y
      (x ⊢> y) [ z ]′ = x ⊢ y [ z ]
```

A.2.4 Display Property

In this section, we will prove that NLQ has the display property. Before we can do this, we will define one more type of context: a display context. This is a context where the inserted structure is always guaranteed to end up at the top-level:

```
data DisplayCtxt : Polarity → Set where
  <⊢_ : Struct − → DisplayCtxt +
  _⊢> : Struct + → DisplayCtxt −

instance
  Pluggable-Display : Pluggable (DisplayCtxt p) (Struct p) Sequent
  Pluggable-Display = record { _[_] = _[_]′ }
    where
      _[_]′ : DisplayCtxt p → Struct p → Sequent
      (<⊢ y) [ x ]′ = x ⊢ y
      (x ⊢>) [ y ]′ = x ⊢ y
```

Now we can defined DP: a type-level function, which takes a sequent context and computes a display context in which the structure that would be in the hole of the sequent context is displayed (i.e. brought to the top-level).

This is implemented with two functions, DPL and DPR, which manipulate the antecedent and succedent. By splitting up the sequent in two arguments—the antecedent and the succedent—these functions become structurally recursive. Note that what these functions encode is basically the relations established by residuation:

```
mutual
  DP : (s : SequentCtxt p) → DisplayCtxt p
  DP (x <⊢ y) = DPL x y
  DP (x ⊢> y) = DPR x y
```

```
DPL : (x : StructCtxt p +) (y : Struct −) → DisplayCtxt p
DPL ( HOLE         ) z = <⊢ z
DPL ( DIA1   k x   ) z = DPL x ( BOX  k z   )
DPL ( PROD1  k x y ) z = DPL x ( IMPL k z y )
DPL ( PROD2  k x y ) z = DPL y ( IMPR k x z )

DPR : (x : Struct +) (y : StructCtxt p −) → DisplayCtxt p
DPR x ( HOLE         ) = x ⊢>
DPR x ( BOX1  k y    ) = DPR ( DIA  k x   ) y
DPR x ( IMPR1 k y z  ) = DPL y ( IMPL k z x )
DPR x ( IMPR2 k y z  ) = DPR ( PROD k y x ) z
DPR x ( IMPL1 k z y  ) = DPR ( PROD k x y ) z
DPR x ( IMPL2 k z y  ) = DPL y ( IMPR k x z )
```

The actual displaying is done by the term-level function dp. This function takes a sequent context s (as above), a structure w, and a proof for the sequent $s[w]$. It then computes an isomorphism between proofs of $s[w]$ and proofs of $\mathsf{DP}(s)[w]$ where, in the second proof, the structure w is guaranteed to be displayed:[1]

```
mutual
  dp : (s : SequentCtxt p) (w : Struct p) → (NLQ s [ w ]) ⇔ (NLQ DP(s)[ w ])
  dp (x <⊢ y) w = dpL x y w
  dp (x ⊢> y) w = dpR x y w

  dpL  : (x : StructCtxt p +) (y : Struct −) (w : Struct p)
       → (NLQ x [ w ] ⊢ y) ⇔ (NLQ DPL x y [ w ])
  dpL ( HOLE         )  z w = I.id
  dpL ( DIA1   k x   )  z w = dpL x ( BOX  k z   )  w I.∘ mkISO resDB resBD
  dpL ( PROD1  k x y )  z w = dpL x ( IMPL k z y )  w I.∘ mkISO resPL resLP
  dpL ( PROD2  k x y )  z w = dpL y ( IMPR k x z )  w I.∘ mkISO resPR resRP

  dpR  : (x : Struct +) (y : StructCtxt p −) (w : Struct p)
       → (NLQ x ⊢ y [ w ]) ⇔ (NLQ DPR x y [ w ])
  dpR x ( HOLE         ) w = I.id
  dpR x ( BOX1  k y    ) w = dpR ( DIA  k x   ) y  w I.∘ mkISO resBD resDB
  dpR x ( IMPR1 k y z  ) w = dpL y ( IMPL k z x )    w I.∘ mkISO resRL resLR
  dpR x ( IMPR2 k y z  ) w = dpR ( PROD k y x ) z  w I.∘ mkISO resRP resPR
  dpR x ( IMPL1 k z y  ) w = dpR ( PROD k x y ) z  w I.∘ mkISO resLP resPL
  dpR x ( IMPL2 k z y  ) w = dpL y ( IMPR k x z )    w I.∘ mkISO resLR resRL
```

Note that while they are defined under a mutual-keyword, these functions are not mutually recursive—however, if the logic NLQ contained e.g. subtractive types as found in LG, they would be.

Below we define dp1 and dp2, which are helper functions. These functions allow you to access the two sides of the isomorphism more easily:

[1] In the definition of dp we use some definitions from the Agda standard library, related to isomorphisms, found under Function.Equivalence. An isomorphism is written ⇔, and created with mkISO—which was renamed from equivalence. Identity and composition are written as usual, with the module prefix I. Application is written with a combination of from/to and ⟨$⟩.

```
dp1 : (s : SequentCtxt p) NLQ s [ w ] → NLQ DP(s)[ w ]
dp1 s f = to (dp s w) ⟨$⟩ f

dp2 : (s : SequentCtxt p) NLQ DP(s)[ w ] → NLQ s [ w ]
dp2 s f = from (dp s w) ⟨$⟩ f
```

A.2.5 Structuralising Types

Because each logical connective has a structural equivalent, it is possible—to a certain extend—structuralise logical connectives en masse. The function St takes a type, and computes the maximally structuralised version of that type, given a target polarity p:

```
St : Type → Struct p
St { p = + } ( Dia   k  a  )  = DIA   k (St a)
St { p = − } ( Box   k  a  )  = BOX   k (St a)
St { p = + } ( UnitL k  a  )  = PROD  k (UNIT k) (St a)
St { p = − } ( ImpR  k  a b )  = IMPR  k (St a) (St b)
St { p = − } ( ImpL  k  b a )  = IMPL  k (St b) (St a)
St { p = _ } a                 = · a ·
```

We know that if we try to structuralise a positive type as a negative structure, or vice versa, it results in the primitive structure. The lemma lem-St encodes this knowledge:

```
lem-St : a → Pol(a) ≡ ~ p → St a ≡ · a ·
lem-St ( El  a       ) pr = refl
lem-St ( El  a       ) pr = refl
lem-St ( Dia   k a   ) ()
lem-St ( Dia   k a   ) pr = refl
lem-St ( Box   k a   ) pr = refl
lem-St ( Box   k a   ) ()
lem-St ( UnitL k a   ) ()
lem-St ( UnitL k a   ) pr = refl
lem-St ( ImpR  k a b ) pr = refl
lem-St ( ImpR  k a b ) ()
lem-St ( ImpL  k b a ) pr = refl
lem-St ( ImpL  k b a ) ()
```

The functions st, stL and stR actually perform the structuralisation described by St. Given a proof for a sequent s, they will structuralise either the antecedent, the succedent, or both:

```
mutual
  st : NLQ St a ⊢ St b → NLQ · a · ⊢ · b ·
  st f = stL (stR f)

  stL : NLQ St a ⊢ y → NLQ · a · ⊢ y
  stL { a = El       a   } f = f
  stL { a = Dia    k  a   } f = diaL (resBD (stL (resDB f)))
```

```
stL { a = Box   k  a   } f = f
stL { a = UnitL k  a   } f = unitLL (resRP (stL (resPR f)))
stL { a = ImpR  k  a b } f = f
stL { a = ImpL  k  b a } f = f

stR : NLQ x ⊢ St b → NLQ x ⊢ · b ·
stR { b = El      a   } f = f
stR { b = Dia   k a   } f = f
stR { b = Box   k a   } f = boxR (resDB (stR (resBD f)))
stR { b = UnitL k a   } f = f
stR { b = ImpR  k a b } f = impRR (resPR (stR (resLP (stL (resPL (resRP f))))))
stR { b = ImpL  k b a } f = impLR (resPL (stR (resRP (stL (resPR (resLP f))))))
```

A.2.6 Identity Expansion

Another important proof is 'identity expansion'—the proof that tells us that although we have restricted the axioms to atomic types, we can still derive the full identity rule. The inclusion of focusing makes this proof slightly more complex, as between the introduction of the connectives, we have to structuralise and occasionally switch focus.

In the below proof, axR and axL recursively apply the rules for symmetric introduction—through axR′ and axL′—until there is a clash in polarity—which is defined as applying axR to a negative type or vice versa—at which point they switch focus, structuralise, and continue:[2]

```
mutual
  ax : NLQ · a · ⊢ · a ·
  ax with Pol(a) | inspect Pol(a)
  ... | + | P.[ p ]  rewrite lem-St  a p  = stL (focR p (axR′ p))
  ... | − | P.[ n ]  rewrite lem-St  a n  = stR (focL n (axL′ n))

  axR : NLQ St b ⊢[ b ]
  axR with Pol(b) | inspect Pol(b)
  ... | + | P.[ p ]                       = axR′ p
  ... | − | P.[ n ]  rewrite lem-St  b n  = unfR n (stR (focL n (axL′ n)))

  axL : NLQ [ a ]⊢ St a
  axL with Pol(a) | inspect Pol(a)
  ... | + | P.[ p ]  rewrite lem-St  a p  = unfL p (stL (focR p (axR′ p)))
  ... | − | P.[ n ]                       = axL′ n

  axR′ : Pol(b) ≡ + → NLQ St b ⊢[ b ]
  axR′ { b = El      a   } p = axElR p
  axR′ { b = Dia   k a   } p = diaR axR
  axR′ { b = Box   k a   } ()
  axR′ { b = UnitL k a   } p = unitLR axR
  axR′ { b = ImpR  k a b } ()
```

[2] In the definition of ax, axR and axL we use inspect, which allows you to apply a function f to an argument x to obtain y, and obtain an explicit proof that $f\ x \equiv y$. The function inspect is defined in Relation.Binary.PropositionalEquality.

```
axR′ { b = ImpL   k  b a  } ()

axL′ : Pol(a) ≡ − → NLQ [ a ]⊢ St a
axL′ { a = El         a    } n = axElL n
axL′ { a = Dia    k   a    } ()
axL′ { a = Box    k   a    } n = boxL axL
axL′ { a = UnitL  k   a    } ()
axL′ { a = ImpR   k   a b  } n = impRL axR axL
axL′ { a = ImpL   k   b a  } n = impLL axR axL
```

A.2.7 Quanfitier Raising

In this section, we show that $\mathbf{Q}\uparrow$ and $\mathbf{Q}\downarrow$ are indeed derivable in the calculus NLQ. For this, we define yet another type of context: the •-Ctxt, i.e. contexts made up solely out of solid products:

```
data •-Ctxt : Set where
  HOLE  : •-Ctxt
  PROD1 : •-Ctxt     → Struct +  → •-Ctxt
  PROD2 : Struct +   → •-Ctxt    → •-Ctxt

instance
  Pluggable-• : Pluggable •-Ctxt (Struct +) (Struct +)
  Pluggable-• = record { _[_] = _[_]′ }
    where
      _[_]′ : •-Ctxt → Struct + → Struct +
      ( HOLE        ) [ z ]′ = z
      ( PROD1  x y  ) [ z ]′ = PROD Solid    (x [ z ]′) y
      ( PROD2  x y  ) [ z ]′ = PROD Solid x  (y [ z ]′)
```

For these contexts, we can define the trace function, which inserts the correct trace of $\mathbf{I}$'s, $\mathbf{B}$'s and $\mathbf{C}$'s:

```
trace : •-Ctxt → Struct +
trace ( HOLE        ) = UNIT (Quan Weak)
trace ( PROD1  x y  ) = PROD Solid (PROD Solid C (trace x)) y
trace ( PROD2  x y  ) = PROD Solid (PROD Solid B x) (trace y)
```

And using the trace function, we can define upwards and downwards quantifier movement:

```
qL : ∀ x → NLQ trace(x) ⊢[ b ] → NLQ [ c ]⊢ y → NLQ x [ · QW (b \\ c) · ] ⊢ y
qL x f g = ↑ x (resRP (focL refl (impRL f g)))
  where
  ↑ : x → NLQ trace(x) ∘ · a · ⊢ z → NLQ x [ · QW a · ] ⊢ z
  ↑ x f = init x (move x f)
    where
    init   : (x : •-Ctxt) → NLQ x [ I ∘ · a · ] ⊢ z → NLQ x [ · QW a · ] ⊢ z
    init   ( HOLE        ) f = unitLL f
    init   ( PROD1  x y  ) f = resLP (init x (resPL f))
```

```
    init    ( PROD2  x y  ) f = resRP (init y (resPR f))
    move  : (x : •-Ctxt) → NLQ trace(x) ∘ y ⊢ z → NLQ x [ I ∘ y ] ⊢ z
    move  ( HOLE        ) f = f
    move  ( PROD1  x y  ) f = resLP (move x (resPL (upC f)))
    move  ( PROD2  x y  ) f = resRP (move y (resPR (upB f)))

qR : ∀ x → NLQ x [ · a · ] ⊢ · b · → NLQ trace(x) ⊢ · b // a ·
qR x f = impLR (resPL (↓ x f))
  where
  ↓ : x → NLQ x [ y ] ⊢ z → NLQ trace(x) ∘ y ⊢ z
  ↓ ( HOLE        ) f = unitLI f
  ↓ ( PROD1  x y  ) f = dnC (resLP (↓ x (resPL f)))
  ↓ ( PROD2  x y  ) f = dnB (resRP (↓ y (resPR f)))
```

These compose to form full quantifier movement:

```
q  : (x : •-Ctxt) → NLQ x [ · a · ] ⊢ · b ·
   → NLQ [ c ]⊢ y
   → NLQ x [ · QW ((b // a) \\ c) · ] ⊢ y
q x f g  = qL x (unfR refl (qR x f)) g
```

A.2.8 Infixation and Reasoning with Gaps

The final type of movement to discuss is the derived version of the R_{rgap} rules used by Barker and Shan (2015). First we will formalise the right infixation, allowing a structure with an infixation licence to move downwards past solid products:

```
extR : (x : •-Ctxt) → NLQ x [ y • ♦↓ z ] ⊢ w → NLQ x [ y ] • ♦↓ z ⊢ w
extR ( HOLE        ) f = f
extR ( PROD1  x y  ) f = extLR (resLP (extR x (resPL f)))
extR ( PROD2  x y  ) f = extRR (resRP (extR y (resPR f)))
```

However, here we run into a slight problem. In this formalisation, we use focusing. However, we do not have a full adaptation of the normalisation procedure from display NLQ to focused NLQ to NLQ. In order to fully encode Barker and Shan's rule, we would have to infixate and then *remove* the license. However, removing the license *in this context* is only possible in the case where the type under the licence is positive. So, without problems, we can define the following version of the rule:

```
r↓⁺  : (x : •-Ctxt) (pr : Pol(b) ≡ +)
     → NLQ x [ y • · b · ] ⊢ · c · →  NLQ x [ y ] ⊢ · c ↓ b ·
r↓⁺ x pr f = impLR (resPL (resRP (diaL (resPR (extR x (stop x f))))))
  where
  stop : (x : •-Ctxt) → NLQ x [ y • · b · ] ⊢ z → NLQ x [ y • ♦↓ · □↓ b · ] ⊢ z
  stop ( HOLE        ) f = resRP (resBD (focL refl (boxL (unfL pr (resPR f)))))
  stop ( PROD1  x y  ) f = resLP (stop x (resPL f))
  stop ( PROD2  x y  ) f = resRP (stop y (resPR f))
```

However, in the case where the type under the licence is negative, we will have to use the following, more general rule which leaves the license in place, and then remove it at a later stage in the proof:

```
r↓ : (x : •-Ctxt) → NLQ x [ y • ♦↓ · □↓ b · ] ⊢ · c · →  NLQ x [ y ] ⊢ · c ↓ b ·
r↓ x f = impLR (resPL (resRP (diaL (resPR (extR x f)))))
```

The proofs for left infixation, and extraction can be done in a similar manner.

A.3 Semantics in Agda

Having formalised the syntactic calculus NLQ in the first part, we will now briefly turn our attention towards a semantics. Instead of formalising $\lambda^{\rightarrow}_{\{\mathbf{e},\mathbf{t}\}}$, we will give the semantics for NLQ in Agda—it looks much nicer, and is much less work, even if λΠ is a little bit more expressive than strictly necessary.

We will give our semantics in a separate module, which will—once again—be abstracting over atomic types and their polarity. In addition to this, we now have to abstract over a translation from atomic types to semantic types. For this, we define the following class of translations:

```
record Translate (T1 : Set t1) (T2 : Set t2) : Set (t1 ⊔ t2) where
  field
    _* : T1 → T2
open Translate {{...}}
```

And abstract accordingly:

```
module Semantics
  (Atom : Set)
  (PolarisedAtom  : Polarised Atom)
  (TranslateAtom  : Translate Atom Set)
  where
```

The translation on types, structures and sequents is rather simple. Instead of translating sequents to sequents, we will translate them to function types. Implications too, both logical and structural, become function types. Otherwise, products become products, units becomes units, etc.

```
  instance
    TranslateType : Translate Type Set
    TranslateType = record { _* = _*′ }
      where
        _*′ : Type → Set
        El      a     *′ = a *
        Dia     _ a   *′ = a *′
        Box     _ a   *′ = a *′
        UnitL   _ a   *′ = a *′
        ImpR    _ a b *′ = a *′ → b *′
        ImpL    _ b a *′ = a *′ → b *′
```

```
TranslateStruct : Translate (Struct p) Set
TranslateStruct = record { _* = _*′ }
  where
    _*′ : Struct p → Set
    · a ·          *′ = a *
    B              *′ = ⊤
    C              *′ = ⊤
    I*             *′ = ⊤
    DIA    _ x     *′ = x *′
    UNIT   _       *′ = ⊤
    PROD   _ x y   *′ = x *′ × y *′
    BOX    _ x     *′ = x *′
    IMPR   _ x y   *′ = x *′ → y *′
    IMPL   _ y x   *′ = x *′ → y *′

TranslateSequent : Translate Sequent Set
TranslateSequent = record { _* = _*′ }
  where
    _*′ : Sequent → Set
    ( x ⊢ y    )*′ = x * → y *
    ( [ a ]⊢ y )*′ = a * → y *
    ( x ⊢[ b ] )*′ = x * → b *
```

Finally, using our translation on sequents, we can implement the translation on proofs:

```
instance
  TranslateProof : Translate (NLQ s) (s *)
  TranslateProof = record { _* = _*′ }
    where
      _*′ : NLQ s → s *
      axElR _     *′ = λ x → x
      axElL _     *′ = λ x → x
      unfR  _ f   *′ = f *′
      unfL  _ f   *′ = f *′
      focR  _ f   *′ = f *′
      focL  _ f   *′ = f *′
      impRL f g   *′ = λ h → g *′ ∘ h ∘ f *′
      impRR f     *′ = f *′
      impLL f g   *′ = λ h → g *′ ∘ h ∘ f *′
      impLR f     *′ = f *′
      resRP f     *′ = λ{ (x , y) → (f *′) y x }
      resLP f     *′ = λ{ (x , y) → (f *′) x y }
      resPR f     *′ = λ{ y x → (f *′) (x , y) }
      resPL f     *′ = λ{ x y → (f *′) (x , y) }
      diaL  f     *′ = f *′
      diaR  f     *′ = f *′
      boxL  f     *′ = f *′
      boxR  f     *′ = f *′
```

```
resBD  f  *′ = f *′
resDB  f  *′ = f *′
unitLL f  *′ = λ{ x           → (f *′) (_ , x)       }
unitLR f  *′ = λ{ (_ , x)     → (f *′) x             }
unitLI f  *′ = λ{ (_ , x)     → (f *′) x             }
dnB    f  *′ = λ{ (((_ , x) , y) , z) → (f *′) (x , (y , z)) }
dnC    f  *′ = λ{ (((_ , x) , z) , y) → (f *′) ((x , y) , z) }
dnI*   f  *′ = λ{ ((_ , x) , y)       → (f *′) (x , y)       }
upB    f  *′ = λ{ (x , y , z)         → (f *′) (((_ , x) , y) , z) }
upC    f  *′ = λ{ ((x , y) , z)       → (f *′) (((_ , x) , z) , y) }
upI*   f  *′ = λ{ (x , y)             → (f *′) ((_ , x) , y) }
ifxRR  f  *′ = λ{ (x , y , z)         → (f *′) ((x , y) , z) }
ifxLR  f  *′ = λ{ ((x , z) , y)       → (f *′) ((x , y) , z) }
ifxLL  f  *′ = λ{ ((z , y) , x)       → (f *′) (z , y , x)   }
ifxRL  f  *′ = λ{ (y , z , x)         → (f *′) (z , y , x)   }
extRR  f  *′ = λ{ ((x , y) , z)       → (f *′) (x , y , z)   }
extLR  f  *′ = λ{ ((x , y) , z)       → (f *′) ((x , z) , y) }
extLL  f  *′ = λ{ (z , y , x)         → (f *′) ((z , y) , x) }
extRL  f  *′ = λ{ (z , y , x)         → (f *′) (y , z , x)   }
```

A.4 Example

We will need some language to describe natural language semantics. Agda has a built-in type for Booleans, but we are not really interested in computing anything, so we will simply postulate everything:

```
postulate
  Entity : Set
  Bool   : Set
  exists : (a → Bool) → Bool
  forAll : (a → Bool) → Bool
  _⊃_    : Bool → Bool → Bool
  _∧_    : Bool → Bool → Bool
```

We define the atomic types, their translation to Agda types, and a concept of polarity:

```
data Atom : Set where S N NP : Atom
```

The translation function then follows, and with it we can instantiate the syntax and semantics modules:

```
TranslateAtom : Translate Atom Set
TranslateAtom = record { _* = _*′ }
  where
    _*′ : Atom → Set
    S  *′ = Bool
    N  *′ = Entity → Bool
    NP *′ = Entity
```

```
PolarisedAtom : Polarised Atom
PolarisedAtom = record { Pol = λ _ → - }
```

Then we define the syntactic types for our example sentence. There are much better ways to do this—building a lexicon, computing the sequent from the given words, etc—and many of these are used in the Haskell implementation, but since the Agda version lacks proof search, there is no real reason to invest in all this machinery:

```
MARY SEES FOXES : Struct +
MARY  = · El NP ·
SEES  = · (El NP \ El S) / El NP ·
FOXES = · QW((El S ⫽ El NP) ⧵ El S) ·
```

A proof for this sentence is easily given:

```
syn0 : NLQ MARY • SEES • FOXES ⊢ · El S ·
syn0 = qL (PROD2 _ (PROD2 _ HOLE)) (unfR refl
     ( qR (PROD2 _ (PROD2 _ HOLE)) (resRP (resLP (focL refl
     ( impLL axR (impRL axR axL))))))) axL
```

We then postulate some primitive meanings, and use these to give some definitions for our lexical entries. The real work is done in the definition of foxes:

```
postulate
  mary : Entity
  see  : Entity → Entity → Bool
  fox  : Entity → Bool

sees : SEES *
sees y x = see x y

foxes : FOXES *
foxes v = exists (λ f → forAll (λ x → f x ⊃ (fox x ∧ v x)))
```

And finally, we translate our syntactic proof, insert the lexical entries, and normalise, et voilà! We have our semantics:

```
sem0 : (syn0 *) (mary , sees , foxes)
     ≡ exists (λ f → forAll (λ x → f x ⊃ (fox x ∧ see mary x)))
sem0 = refl
```

A.5 CPS-semantics and indefinites

We mentioned that one possible way of dealing with indefinites is to extend the CPS-semantics for focused NL, given earlier, to full NLQ. This would allow us to model quantifiers using the **IBC**-rules, but indefinites using a semantic CPS-translation. In order to do this, we are going to define the CPS-semantics for NLQ. We abstract in our module header, much like we did for our Semantics module:

```
module CPS-Semantics
  (Atom : Set) (R : Set)
  (PolarisedAtom  : Polarised Atom)
  (TranslateAtom  : Translate Atom Set)
  where
```

And we define some convenient syntax for continuation types:

```
_ R : Set → Set
a R = a → R
```

The most complex part of the CPS-translation is the polarity-driven translation on types, formalised below:

```
⟦_⟧_ : Type → Polarity → Set
⟦ El  a       ⟧ + with Pol(a)
⟦ El  a       ⟧ + | +  = a *
⟦ El  a       ⟧ + | −  = a * R R
⟦ Dia   k a   ⟧ +      = ⟦ a ⟧ +
⟦ Box   k a   ⟧ +      = ⟦ a ⟧ − R
⟦ UnitL k a   ⟧ +      = ⟦ a ⟧ +
⟦ ImpR  k a b ⟧ +      = ⟦ a ⟧ + × ⟦ b ⟧ − R
⟦ ImpL  k b a ⟧ +      = ⟦ b ⟧ − × ⟦ a ⟧ + R
⟦ El  a       ⟧ −      = a * R
⟦ Dia   k a   ⟧ −      = ⟦ a ⟧ + R
⟦ Box   k a   ⟧ −      = ⟦ a ⟧ −
⟦ UnitL k a   ⟧ −      = ⟦ a ⟧ + R
⟦ ImpR  k a b ⟧ −      = ⟦ a ⟧ + × ⟦ b ⟧ −
⟦ ImpL  k b a ⟧ −      = ⟦ b ⟧ − × ⟦ a ⟧ +
```

For structures and sequents, the translations are simple, and we can resort to using our previous Translate class. As mentioned, we simply translate *all* structural connectives as products:

```
instance
  TranslateStruct : Translate (Struct p) Set
  TranslateStruct = record { _* = _*′ }
    where
    _*′ : Struct p → Set
    _*′ · a · = ⟦ a ⟧ p
    B             *′ = ⊤
    C             *′ = ⊤
    I*            *′ = ⊤
    DIA   k x     *′ = x *′
    UNIT  k       *′ = ⊤
    PROD  k x y   *′ = x *′ × y *′
    BOX   k x     *′ = x *′
    IMPR  k x y   *′ = x *′ × y *′
    IMPL  k y x   *′ = y *′ × x *′
```

And we translate sequents as Agda functions:

```
instance
  TranslateSequent : Translate Sequent Set
  TranslateSequent = record { _* = _*′ }
    where
    _*′ : Sequent → Set
    (  x ⊢  y  ) *′ = x * → y * → R
    ([ a ]⊢  y  ) *′ = y * → ⟦ a ⟧ −
    (  x ⊢[ b ]) *′ = x * → ⟦ b ⟧ +
```

The final part is the translation on proofs. Before we give the full translation on proofs, we will demonstrate that for all a, if there is a clash in polarity between the polarity of a type and the polarity of the translation, we obtain a "continuation type" $a\ ^R$:

```
lem-⟦·⟧ : (a : Type) → Pol(a) ≡ p → (⟦ a ⟧ ~ p) ≡ (⟦ a ⟧ p R)
lem-⟦·⟧ ( El  a          ) pr rewrite pr = refl
lem-⟦·⟧ ( El  a          ) pr rewrite pr = refl
lem-⟦·⟧ ( Dia    k a     ) pr = refl
lem-⟦·⟧ ( Dia    k a     ) ()
lem-⟦·⟧ ( Box    k a     ) ()
lem-⟦·⟧ ( Box    k a     ) pr = refl
lem-⟦·⟧ ( UnitL  k a     ) pr = refl
lem-⟦·⟧ ( UnitL  k a     ) ()
lem-⟦·⟧ ( ImpR   k a b   ) ()
lem-⟦·⟧ ( ImpR   k a b   ) pr = refl
lem-⟦·⟧ ( ImpL   k a b   ) ()
lem-⟦·⟧ ( ImpL   k a b   ) pr = refl
```

All rules translate to permutations on product types, insert units or map functions over product types. The actual applications and abstractions are hiding in unfR, unfL, focR and focL, which correspond to the translations of the rules of the same name given earlier:

```
instance
  TranslateProof : Translate (NLQ s) (s *)
  TranslateProof = record { _* = _*′ }
    where
    _*′ : NLQ s → s *
    axElR _      *′ = λ x → x
    axElL _      *′ = λ x → x
    unfR  {b = b} n  f  *′ rewrite lem-⟦·⟧ b n = λ x y → (f *′) x y
    unfL  {a = a} p  f  *′ rewrite lem-⟦·⟧ a p = λ y x → (f *′) x y
    focR  {b = b} p  f  *′ rewrite lem-⟦·⟧ b p = λ x k → k ((f *′) x)
    focL  {a = a} n  f  *′ rewrite lem-⟦·⟧ a n = λ k x → k ((f *′) x)
    impRL  f g  *′ = λ{(x , y) → ((f *′) x , (g *′) y)}
    impRR  f    *′ = (f *′)
    impLL  f g  *′ = λ{(x , y) → ((g *′) x , (f *′) y)}
```

```
impLR  f   *′ = (f *′)
resRP  f   *′ = λ{(x , y) z → (f *′) y (x , z)}
resPR  f   *′ = λ{y (x , z) → (f *′) (x , y) z}
resLP  f   *′ = λ{(x , y) z → (f *′) x (z , y)}
resPL  f   *′ = λ{x (z , y) → (f *′) (x , y) z}
diaL   f   *′ = f *′
diaR   f   *′ = f *′
boxL   f   *′ = f *′
boxR   f   *′ = f *′
resBD  f   *′ = f *′
resDB  f   *′ = f *′
unitLL f   *′ = λ{ x → (f *′) (tt , x) }
unitLR f   *′ = λ{ (tt , x) → (f *′) x }
unitLI f   *′ = λ{ (tt , x) → (f *′) x }
dnB    f   *′ = λ{ (((tt , x) , y) , z) → (f *′) (x , (y , z)) }
upB    f   *′ = λ{ (x , (y , z)) → (f *′) (((tt , x) , y) , z) }
dnC    f   *′ = λ{ (((tt , x) , z) , y) → (f *′) ((x , y) , z) }
upC    f   *′ = λ{ ((x , y) , z) → (f *′) (((tt , x) , z) , y) }
upI*   f   *′ = λ{ (x , y) → (f *′) ((tt , x) , y) }
dnI*   f   *′ = λ{ ((tt , x) , y) → (f *′) (x , y) }
ifxRR  f   *′ = λ{ (x , (y , z)) → (f *′) ((x , y) , z) }
ifxLR  f   *′ = λ{ ((x , z) , y) → (f *′) ((x , y) , z) }
ifxLL  f   *′ = λ{ ((z , y) , x) → (f *′) (z , (y , x)) }
ifxRL  f   *′ = λ{ (y , (z , x)) → (f *′) (z , (y , x)) }
extRR  f   *′ = λ{ ((x , y) , z) → (f *′) (x , (y , z)) }
extLR  f   *′ = λ{ ((x , y) , z) → (f *′) ((x , z) , y) }
extLL  f   *′ = λ{ (z , (y , x)) → (f *′) ((z , y) , x) }
extRL  f   *′ = λ{ (z , (y , x)) → (f *′) (y , (z , x)) }
```

Once again, we demonstrate a small example. In this case, the example sentence will be "Everyone said some guest left". This sentence should have an ambiguous interpretation. We will use a CPS-translation for the indefinite "some" to obtain this ambiguity, counting on the scope ambiguity which we can obtain by setting the following polarities in focused NL:

```
PolarisedAtom : Polarised Atom
PolarisedAtom = record { Pol = Pol′ }
  where
    Pol′ : Atom → Polarity
    Pol′ S    = −
    Pol′ N    = +
    Pol′ NP   = +
```

We then define some types for the words in our sentence: we define "everyone" as a syntactic quantifier, but define "someone" as a semantic, CPS-translated quantifier:

```
EVERYONE : Struct +
EVERYONE = · QW((El S ⫽ El NP) ⧵ El S) ·
SAID     : Struct +
```

```
SAID      = · (El NP \ El S) / (◊ El S) ·
SOME      : Struct +
SOME      = · El NP / El N ·
GUEST     : Struct +
GUEST     = · El N ·
LEFT      : Struct +
LEFT      = · El NP \ El S ·
```

The result is that, while "some" cannot escape the scope island through syntactic movement, it nonetheless takes scope, through the CPS-translation. And, because of our chosen polarisation, there are three different derivations: in syn1a, we start by collapsing "some guest", then let "everyone" take scope, and only then collapse the sentence scope and resolve the embedded clause; in syn1b, we start by letting "everyone"' take scope, then we collapse "some guest"', and again end by collapsing the sentence scope and resolving the embedded clause; and, in syn1c, we again start by letting "everyone" take scope, but this time we collapse the sentence scope, and move to the embedded clause, *before* we collapse "some person".

```
syn1a : NLQ EVERYONE • SAID • ⟨ ( SOME • GUEST ) • LEFT ⟩ ⊢ · El S ·
syn1a = (dp2 ((_ •> (_ •> (⧫> ((HOLE <• _) <• _)))) <⊢ _)
         (focL refl (impLL axR (unfL refl
         (dp1 ((_ •> (_ •> (⧫> (HOLE <• _)))) <⊢ _)
         (flip (q (PROD1 HOLE _)) axL
         (dp2 ((_ •> (HOLE <• _)) <⊢ _)
         (focL refl (impLL (diaR (unfR refl (resRP (focL refl axL)))) axL
         )))))))))

syn1b : NLQ EVERYONE • SAID • ⟨ ( SOME • GUEST ) • LEFT ⟩ ⊢ · El S ·
syn1b = (flip (q (PROD1 HOLE _)) axL
         (dp2 ((_ •> (_ •> (⧫> ((HOLE <• _) <• _)))) <⊢ _)
         (focL refl (impLL axR (unfL refl
         (dp1 ((_ •> (_ •> (⧫> (HOLE <• _)))) <⊢ _)
         (dp2 ((_ •> (HOLE <• _)) <⊢ _)
         (focL refl (impLL (diaR (unfR refl (resRP (focL refl axL)))) axL
         )))))))))

syn1c : NLQ EVERYONE • SAID • ⟨ ( SOME • GUEST ) • LEFT ⟩ ⊢ · El S ·
syn1c = (flip (q (PROD1 HOLE _)) axL
         (dp2 ((_ •> (HOLE <• _)) <⊢ _)
         (focL refl (flip impLL axL (diaR (unfR refl
         (dp2 (((HOLE <• _) <• _) <⊢ _)
         (focL refl (impLL axR (unfL refl (resPL (resRP (focL refl axL)
         ))))))))))))
```

In order to assign an interpretation to our derivations, we give some definitions for our lexical terms. These are now slightly more complicated, using the CPS-translated types:

```
postulate
```

```
person  : Entity → Bool
guest   : Entity → Bool
say     : Entity → Bool → Bool
leave   : Entity → Bool

everyone          : EVERYONE *
everyone (f , k)  = forAll (λ x → person x ⊃ (f (k , x)))
said              : SAID *
said ((x , k) , k')  = k (say x (k' (λ y → y)))
some              : SOME *
some (k , f)      = exists (λ x → f x ∧ k x)
left              : LEFT *
left (x , k)      = k (leave x)
```

And, when we compute our meanings—inserting the identity function in order to extract a meaning out of the continuation—we see that we get exactly the desired result:

```
sem1a  : (syn1a *) (everyone , said , (some , guest) , left) (λ x → x)
       ≡ exists (λ y → guest y ∧ forAll (λ x → person x ⊃ say x (leave y)))
sem1a  = refl
sem1b  : (syn1b *) (everyone , said , (some , guest) , left) (λ x → x)
       ≡ forAll (λ x → person x ⊃ exists (λ y → guest y ∧ say x (leave y)))
sem1b  = refl
sem1c  : (syn1c *) (everyone , said , (some , guest) , left) (λ x → x)
       ≡ forAll (λ x → person x ⊃ (say x (exists (λ y → guest y ∧ leave y))))
sem1c  = refl
```

B Formalisation of NLQ in Haskell

In this appendix, we will discuss the Haskell 'formalisation' of NLQ. Now, formalisation is a bit of an overstatement when talking about Haskell, since the Haskell type-system itself is unsound, as the language allows non-termination. However, in modern Haskell, using the singletons library, it is quite possible to do verification.

The reason that we have implemented a Haskell version as well as an Agda version of NLQ, is because while Agda excels at verification, it is quite a slow language, and we want to do proof search. The Haskell version has very good performance when it comes to proof search. However, writing proofs in Haskell is tedious, and in some instances (mostly when contexts are used) the Haskell version of an Agda proof will require the use of *unsafeCoerce*—though all instances of *unsafeCoerce* can be removed when *InjectiveTypeFamilies* are released.

Because the two implementations are so similar, and because discussing a proof search algorithm is rather boring, in what follows we will *only* discuss the interface to NLQ—whatever you need to write your own examples. The first step in this, is to import the NLQ prelude.

import *NLQ.Prelude*

Once we've done this, we can start the lexicon.

B.1 The Lexicon

The main function to construct lexical entries, is the function *lex*, which has the following type:[1]

```
lex :: (SingI a) => Name -> Word a
```

That is to say, it takes a *Name*—which is just a string—and returns a *Word* of type *a*. However, since the type *a* only occurs in the result-type, it important to always explicitly write the type. For instance, below we define a small lexicon of proper nouns, verbs and nouns:

john, *mary*, *bill*, *alice* :: *Word NP*
john = *lex* `"john"`
mary = *lex* `"mary"`
bill = *lex* `"bill"`
alice = *lex* `"alice"`

run, *leave* :: *Word IV*
run = *lex* `"run"` ; *runs* = *run*
leave = *lex* `"leave"` ; *leaves* = *leave*

read, *see*, *like*, *serve*, *fear*, *know* :: *Word TV*
read = *flip* ◁ *lex* `"read"` ; *reads* = *read*
see = *flip* ◁ *lex* `"see"` ; *sees* = *see*
like = *flip* ◁ *lex* `"like"` ; *likes* = *like*
serve = *flip* ◁ *lex* `"serve"` ; *serves* = *serve*

[1] The *SingI* typeclass is the class of types which have a singleton associated with them—a term-level representation of a type, or a type-level representation for a term, depending on your perspective. All syntactic types have singleton instances associated with them.

$$
\begin{array}{lll}
\mathit{fear} & = \mathit{flip} \triangleleft \mathit{lex}\ \texttt{"fear"} & ; \mathit{fears} \quad = \mathit{fear} \\
\mathit{know} & = \mathit{flip} \triangleleft \mathit{lex}\ \texttt{"know"} & ; \mathit{knows} \quad = \mathit{know}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{person}, \mathit{waiter}, \mathit{fox}, \mathit{book}, \mathit{author}, \mathit{ocean} :: \mathit{Word}\ N & & \\
\mathit{person} & = \mathit{lex}\ \texttt{"person"} & ; \mathit{people} \quad = \mathit{plural} \triangleleft \mathit{person} \\
\mathit{waiter} & = \mathit{lex}\ \texttt{"waiter"} & ; \mathit{waiters} \quad = \mathit{plural} \triangleleft \mathit{waiter} \\
\mathit{fox} & = \mathit{lex}\ \texttt{"fox"} & ; \mathit{foxes} \quad = \mathit{plural} \triangleleft \mathit{fox} \\
\mathit{book} & = \mathit{lex}\ \texttt{"book"} & ; \mathit{books} \quad = \mathit{plural} \triangleleft \mathit{book} \\
\mathit{author} & = \mathit{lex}\ \texttt{"author"} & ; \mathit{authors} \quad = \mathit{plural} \triangleleft \mathit{author} \\
\mathit{ocean} & = \mathit{lex}\ \texttt{"ocean"} & ; \mathit{oceans} \quad = \mathit{plural} \triangleleft \mathit{ocean} \\
\mathit{country} & = \mathit{lex}\ \texttt{"country"} & ; \mathit{countries} = \mathit{plural} \triangleleft \mathit{country}
\end{array}
$$

If you ever get an error like the error given below, you have probably forgotten to give an explicit type for a lexicon entry:

```
NLQ_Haskell.lhs:134:15:
  No instance for (Data.Singletons.SingI a0)
    arising from a use of ‘lex’
  The type variable ‘a0’ is ambiguous
  ...
  In the expression: lex "alice"
  In an equation for ‘alice’: alice = lex "alice"
```

At this point, it is best to ignore the *plural* definitions and the *flip* function; they will make sense soon enough.

Most *interesting* lexical entries are given as *terms* instead of simply as postulates. For this, there is the function $\mathit{lex\_}$, which has the following type:

```
lex_ :: (SingI a) => Hask (H a) -> Word a
```

Here, H is a function which translates syntactic types to semantic types, and *Hask* is a function which translates semantic types to Haskell types. In addition, *Hask* prefixes all atomic types with the datatype *Expr*, which allows you to use "postulates"—more on this below. So, if you write, e.g. a lexical entry of type N using $\mathit{lex\_}$, you will be expected to provide something of type $\mathit{Expr}\ \mathbf{e} \to \mathit{Expr}\ \mathbf{t}$:

$$
\begin{array}{l}
a, \mathit{some} :: \mathit{Word}\ (\mathbf{Q}^S((S /\!\!/ NP) \setminus\!\!\setminus S)/N) \\
a \quad = \mathit{some} \\
\mathit{some} = \mathit{lex\_}\ (\lambda f\ g \to \exists_{\mathbf{e}}\ (\lambda x \to f\ x \wedge g\ x))
\end{array}
$$

$$
\begin{array}{l}
\mathit{no}, \mathit{every} :: \mathit{Word}\ (\mathbf{Q}^W((S /\!\!/ NP) \setminus\!\!\setminus S)/N) \\
\mathit{no} \quad = \mathit{lex\_}\ (\lambda f\ g \to \neg\ (\exists_{\mathbf{e}}\ (\lambda x \to f\ x \wedge g\ x))) \\
\mathit{every} = \mathit{lex\_}\ (\lambda f\ g \to \forall_{\mathbf{e}}\ (\lambda x \to f\ x \supset g\ x))
\end{array}
$$

The functions $\forall_{\mathbf{e}}$, $\exists_{\mathbf{e}}$, $\wedge$ and $\supset$ are defined in *NLQ.Prelude*.

Now that we have definitions for *some*, *every*, and *person* it would be a shame if we could not simply give definitions for somebody, everybody, and so on, by applying the one to the other. However, since words are directionally typed, we cannot use Haskell's function application. Instead, *NLQ.Prelude* provides you with two directional versions of application, written $\triangleleft$ and $\triangleright$:

$$
\begin{array}{lll}
\mathit{somebody} = \mathit{some} & \triangleleft \mathit{person}; \mathit{someone} & = \mathit{somebody} \\
\mathit{nobody} \quad = \mathit{no} & \triangleleft \mathit{person}; \mathit{noone} & = \mathit{nobody}
\end{array}
$$

$everybody = every \triangleleft person; everyone = everybody$

This is the same $\triangleleft$ that was used in the definitions of the plural nouns above. In fact, *plural* is simply a word—albeit a somewhat complex one:

$$
\begin{array}{l}
plural :: Word\ (NS/N) \\
plural = lex\_ \\
\quad (\lambda f\ g \to \exists_{\mathbf{et}}\ (\lambda h \to moreThanOne\ h \wedge \forall_{\mathbf{e}}\ (\lambda x \to h\ x \supset (f\ x \wedge g\ x)))) \\
\quad \textbf{where} \\
\qquad moreThanOne :: (Expr\ \mathbf{e} \to Expr\ \mathbf{t}) \to Expr\ \mathbf{t} \\
\qquad moreThanOne\ f = \exists_{\mathbf{e}}\ (\lambda x \to \exists_{\mathbf{e}}\ (\lambda y \to f\ x \wedge f\ y \wedge x\ \ y))
\end{array}
$$

Using the same technique, we can define a simple function which flips the arguments on transitive verbs to make the semantics more satisfactory:

$$
\begin{array}{l}
flip :: Word\ (TV/TV) \\
flip = lex\_\ (\lambda tv\ y\ x \to tv\ x\ y)
\end{array}
$$

Note that the definition of *plural* uses an existential quantifying over *predicates*. In fact, our semantic calculus is a higher-order logic, so $\forall_x$ and $\exists_x$ take an extra argument to determine which type they quantify over. This type has to be provided using a singleton. There are a number of predefined singletons for semantic types, amongst which are **e**, **t**, **et**, **eet**, **ett**, **tet**, **tt**, **ttt** and (**et**)**t**, but if you need to form your own, you can use $:\!\to$ to create function types.

Right, let's carry on, and define some simple function words. Note that we are defining *the* as a postulate, since we do not really want to commit to an implementation yet—though if we would want to, it could easily be implemented it using quantification:

$$
\begin{array}{lll}
to & = lex\_\ (\lambda x \to x) & :: Word\ (INF/IV) \\
of & = lex\ \texttt{"of"} & :: Word\ ((N\backslash N)/NP) \\
the & = lex\ \texttt{"the"} & :: Word\ (NP/N)
\end{array}
$$

Next up, *want*. As discussed, *want* has an ambiguous type: it can take either an NP object, a VP object or both! For these examples, we will only use the last two of these meanings:

$$
\begin{array}{l}
want :: Word\ ((IV/INF)\ \&\ ((IV/INF)/NP)) \\
want\ \ = lex\_\ ((want2, want3)) \\
\quad \textbf{where} \\
\quad want2\quad f\ x = (\texttt{"want"} :: \mathbf{ett})\ x\ (f\ x) \\
\quad want3\ y\ f\ x = (\texttt{"want"} :: \mathbf{ett})\ x\ (f\ y) \\
wants = want
\end{array}
$$

Note that in the definition of *want*, we had to have a postulate want, which we called *want*. For this, we have the ::-operator. This operator takes a string and the singleton for a semantic type a, and returns an expression of type $Hask\ a$.

Using the same operator, and the delimiter $\Diamond^W$, we can give a definition of *say*:

$say :: Word\ (IV/\Diamond^W S)$
$say = lex\_\ (\lambda y\ x \to (\texttt{"say"} :: \mathbf{ett})\ x\ y); says = say$

Next, we define the double/parasitic quantifiers *same* and *different*. Note that the function used in these definitions is predefined, as is its negation:

$same, different :: Word\ (\mathbf{Q}^W((S /\!\!/ (\mathbf{Q}^W(((S\ /\!\!/ NP) /\!\!/ A) \setminus\!\!\setminus (S\ /\!\!/ NP)))) \setminus\!\!\setminus S))$
$same = lex\_\ same'$
where
$same'\ k = \exists_{\mathbf{e}}\ (\lambda z \to k\ (\lambda k'\ x \to k'\ (\lambda f\ y \to f\ y :\ y\ z)\ x))$
$different = lex\_\ diff1$
where
$diff1\ k = \exists_{\mathbf{eet}}\ (\lambda f \to diff2\ f \wedge k\ (\lambda k\ x \to k\ (\lambda g\ y \to g\ y \wedge f\ x\ y)\ x))$
$diff2\ f = \forall_{\mathbf{e}}\ (\lambda x \to \forall_{\mathbf{e}}\ (\lambda y \to \neg\ (\exists_{\mathbf{e}}\ (\lambda z \to f\ z\ x \wedge f\ z\ y))))$

Lastly, we define *which*. As an example of the fact that is is plain Haskell, we use **type**-declarations to split up the type for *which* in its two possible types, i.e. whether its going to insert its clause into a right or a left gap. The semantics, however, stay the same:

$which :: Word\ (\mathbf{Q}^W((NP /\!\!/ NP) \setminus\!\!\setminus (((N\backslash N)/(NP\backslash S)) \,\&\, ((N\backslash N)/(S \downharpoonright NP)))))$
$which = lex\_\ (\lambda f \to (\lambda g\ h\ x \to h\ x \wedge (g\ (f\ x)), \lambda g\ h\ x \to h\ x \wedge (g\ (f\ x))))$

And with the lexicon over with, we can start writing examples!

B.2 Examples

In this section, we will treat a number of examples. There are some things that need explaining.

First, the quasiquoter *nlq*. This is not a terribly interesting function, as it does none of the parsing. What it does is take a parse tree, turn it into a tree of Haskell pairs, and feed it to the actual parser, *parseBwd*. For instance, in the first example, "john runs" is taken, and turned into the Haskell expression $parseBwd\ S\ (john, runs)$. The input string is taken as a right-branching parse tree, so any left branches have to be explicitly written. In addition, *weak* delimiters are written as < ... >, and *strong* delimiters are written as << ... >>.

Secondly, with some lhs2TEX magic, I have inserted the results of the examples in listings below them. These listings list the precise output of the Haskell program. What follows is a list of examples, and their interpretations:

$s0 = [nlq \mid john\ runs \mid]$

```
run john
```

$s1 = [nlq \mid john\ likes\ mary \mid]$

```
like john mary
```

$s2 = [nlq \mid someone\ likes\ mary \mid]$

```
∃x0.person x0 ∧ like x0 mary
```

$s3 = [nlq \mid john\ likes\ everyone \mid]$

```
∀x0.person x0 ⊃ like john x0
```

s4 = [*nlq* | *someone likes everyone* |]

```
∃x0.person x0 ∧ (∀x1.person x1 ⊃ like x0 x1)
∀x0.person x0 ⊃ (∃x1.person x1 ∧ like x1 x0)
```

s5 = [*nlq* | (*the waiter*) *serves everyone* |]

```
∀x0.person x0 ⊃ serve (the (λx1.(waiter x1))) x0
```

s6 = [*nlq* | (*the same waiter*) *serves everyone* |]

```
∃x0.(∀x1.person x1 ⊃ serve (the (λx2.(waiter x2 ∧ x2 ≡
    x0))) x1)
```

s7 = [*nlq* | (*a different waiter*) *serves everyone* |]

```
∃x0.(∀x1.(∀x2.(∄x3.x0 x3 x1 ∧ x0 x3 x2))) ∧ (∀x4.person x4 ⊃
     (∃x5.(waiter x5 ∧ x0 x4 x5) ∧ serve x5 x4))
∃x0.(∀x1.(∀x2.(∄x3.x0 x3 x1 ∧ x0 x3 x2))) ∧ (∀x4.person x4 ⊃
     (∃x5.(waiter x5 ∧ x0 x4 x5) ∧ serve x5 x4))
```

s8 = [*nlq* | *mary wants to leave* |]

```
want mary (leave mary)
```

s9 = [*nlq* | *mary* (*wants john*) *to leave* |]

```
want mary (leave john)
```

s10 = [*nlq* | *mary* (*wants everyone*) *to leave* |]

```
∀x0.person x0 ⊃ want mary (leave x0)
```

s11 = [*nlq* | *mary wants to like bill* |]

```
want mary (like mary bill)
```

s12 = [*nlq* | *mary* (*wants john*) *to like bill* |]

```
want mary (like john bill)
```

s13 = [*nlq* | *mary* (*wants everyone*) *to like bill* |]

```
∀x0.person x0 ⊃ want mary (like x0 bill)
```

s14 = [*nlq* | *mary wants to like someone* |]

```
want mary (∃x0.person x0 ∧ like mary x0)
∃x0.person x0 ∧ want mary (like mary x0)
```

s15 = [*nlq* | *mary* (*wants john*) *to like someone* |]

```
want mary (∃x0.person x0 ∧ like john x0)
∃x0.person x0 ∧ want mary (like john x0)
```

s16 = [*nlq* | *mary* (*wants everyone*) *to like someone* |]

```
∀x0.person x0 ⊃ want mary (∃x1.person x1 ∧ like x0 x1)
∀x0.person x0 ⊃ (∃x1.person x1 ∧ want mary (like x0 x1))
∃x0.person x0 ∧ (∀x1.person x1 ⊃ want mary (like x1 x0))
```

s17 = [*nlq* | *mary says* < *john likes bill*> |]

```
say mary (like john bill)
```

s18 = [*nlq* | *mary says* < *everyone likes bill*> |]

```
say mary (∀x0.person x0 ⊃ like x0 bill)
```

s19 = [*nlq* | *mary says* < *someone likes bill*> |]

```
say mary (∃x0.person x0 ∧ like x0 bill)
∃x0.person x0 ∧ say mary (like x0 bill)
```

s20 = [*nlq* | *everyone says* < *someone likes bill*> |]

```
∀x0.person x0 ⊃ say x0 (∃x1.person x1 ∧ like x1 bill)
∀x0.person x0 ⊃ (∃x1.person x1 ∧ say x0 (like x1 bill))
∃x0.person x0 ∧ (∀x1.person x1 ⊃ say x1 (like x0 bill))
```

s21 = [*nlq* | *mary reads a book which john likes* |]

```
∃x0.(book x0 ∧ like john x0) ∧ read mary x0
```

s22 = [*nlq* | *mary reads a book* (*the author of which*) *john likes* |]

```
∃x0.(book x0 ∧ like john (the (λx1.(of x0 (λx2.(author x2))
    x1)))) ∧ read mary x0
```

s23 = [*nlq* | *mary sees foxes* |]

```
∃x0.(∃x1.(∃x2.x0 x1 ∧ x0 x2 ∧ x1 ≢ x2)) ∧ (∀x3.x0 x3 ⊃ (fox
    x3 ∧ see mary x3))
```

s24 = [*nlq* | *mary sees the fox* |]

```
see mary (the (λx0.(fox x0)))
```

s25 = [*nlq* | *mary sees a fox* |]

```
∃x0.fox x0 ∧ see mary x0
```

s26 = [*nlq* | *alice reads a book* (*the author of which*) *fears the ocean* |]

```
∃x0.(book x0 ∧ fear (the (λx1.(of x0 (λx2.(author x2))
    x1))) (the (λx3.(ocean x3)))) ∧ read alice x0
```