Formalising type-logical grammars in Agda

Pepijn Kokke

Utrecht University

Abstract. In recent years, the interest in using proof assistants to formalise and reason about mathematics and programming languages has grown. Type-logical grammars, being closely related to type theories and systems used in functional programming, are a perfect candidate to next apply this curiosity to. The advantages of using proof assistants is that they allow one to write formally verified proofs about one's type-logical systems, and that any theory, once implemented, can immediately be computed with. The downside is that in many cases the formal proofs are written as an afterthought, are incomplete, or use obtuse syntax. This makes it that the verified proofs are often much more difficult to read than the pen-and-paper proofs, and almost never directly published. In this paper, we will try to remedy that by example.
Concretely, we use Agda to model the Lambek-Grishin calculus, a grammar logic with a rich vocabulary of type-forming operations. We then present a verified procedure for cut elimination in this system. Then we briefly outline a continuation-passing style translation from proofs in the Lambek-Grishin calculus to programs in Agda. And finally, we will put our system to use in the analysis of a simple example sentence.

1 Introduction

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 procedure for cut elimination in this paper to investigate what kind of derivations are created by it *and* be confident in its correctness.

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 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.

Additionally, because there is a one-to-one 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.

Our test case in this paper is the Lambek-Grishin calculus (LG, **?**). LG is an example of an extended Lambek calculus. In addition to the product ($\otimes$) and the residual slashes ($\backslash$, $/$), LG has a dual family with $\oplus$ and difference operations ($\oslash$, $\obslash$) together with distributivity principles for the interaction between the two families. See **?** for discussion of how LG overcomes syntactic and semantic limitations of the original Lambek calculus.

We will formalise the residuation-monotonicity axiomatisation for the Lambek-Grishin calculus (**?**) in Agda, present a verified procedure for cut elimination in this system, and briefly outline a continuation-passing style (CPS) translation into Agda. There are several reasons why we have chosen to formalise this particular system.

- It allows cut as an admissible rule, i.e. a function on proofs, instead of defining a separate cut-free system and a cut-elimination procedure;
- it has efficiently decidable proof search, largely due to the absence of the cut rule;
- it has some interesting symmetries, as explored in **??**. Because of this, most proofs of properties of LG are not much more complicated than their associated proofs for the non-associative Lambek calculus;
- it has a continuation-passing style interpretation, which has shown to be useful in both derivational and lexical semantics (**???**);
- lastly, an implementation of the non-associative Lambek calculus can easily and mechanically be extracted from our implementation of LG.

Since this paper is by no means a complete introduction to Agda or to dependently-typed programming, we advise the interested reader to refer to **?** for a detailed discussion of Agda.

It should be mentioned that (although we omit some of the more tedious parts) this paper is written in literate Agda, and the code has been made available on GitHub.[1]

[1] See `https://gist.github.com/pepijnkokke/cc12b92a8a60696b712c#file-main-agda`.

2 Formulas, Judgements, Base System

If we want to model our type-logical grammars in Agda, a natural starting point would be our atomic formulas—such as n, np, s, etc. These could easily be represented as an enumerated data type. However, in order to avoid committing to a certain set of atomic formulas and side-step the debate on which formulas *should* be atomic, we will simply assume there is a some data type representing our atomic formulas. This will be reflected in our module header as follows:

```
module logic (Atom : Set) where
```

Our formulas can easily be described as a data type, injecting our atomic formulas by means of the constructor el, and adding the familiar connectives from the Lambek-Grishin calculus as binary constructors. Note that, in Agda, we can use underscores in definitions to denote argument positions. This means that _⊗_ below defines an infix, binary connective:

```
data Type : Set where
  el                : Atom → Type
  _⊗_ _↖_ _↗_ : Type → Type → Type
  _⊕_ _⊘_ _⊖_ : Type → Type → Type
```

In the same manner, we can define a data type to represent judgements:

```
data Judgement : Set where
  _⊢_ : Type → Type → Judgement
```

Using the above definitions, we can now write judgements such as A ⊗ A ↖ B ⊢ B as Agda values. Next we will define a data type to represent our logical system. This is where we can use the dependent type system to our advantage. The constructors for our data type will represent the axiomatic inference rules of the system, and their *types* will be constrained by judgements. Below you can see the entire system LG as an Agda data type[2]:

```
data LG_ : Judgement → Set where
  ax    : LG el A ⊢ el A
    -- residuation and monotonicity for (↗ , ⊗ , ↖)
  r↖⊗ : LG B ⊢ A ↖ C → LG A ⊗ B ⊢ C
  r⊗↖ : LG A ⊗ B ⊢ C → LG B ⊢ A ↖ C
  r↗⊗ : LG A ⊢ C ↗ B → LG A ⊗ B ⊢ C
  r⊗↗ : LG A ⊗ B ⊢ C → LG A ⊢ C ↗ B
  m⊗  : LG A ⊢ B → LG C ⊢ D → LG A ⊗ C ⊢ B ⊗ D
  m↖  : LG A ⊢ B → LG C ⊢ D → LG B ↖ C ⊢ A ↖ D
  m↗  : LG A ⊢ B → LG C ⊢ D → LG A ↗ D ⊢ B ↗ C
    -- residuation and monotonicity for (⊘ , ⊕ , ⊖)
  r⊖⊕ : LG B ⊖ C ⊢ A → LG C ⊢ B ⊕ A
  r⊕⊖ : LG C ⊢ B ⊕ A → LG B ⊖ C ⊢ A
  r⊕⊘ : LG C ⊢ B ⊕ A → LG C ⊘ A ⊢ B
  r⊘⊕ : LG C ⊘ A ⊢ B → LG C ⊢ B ⊕ A
  m⊕  : LG A ⊢ B → LG C ⊢ D → LG A ⊕ C ⊢ B ⊕ D
  m⊖  : LG C ⊢ D → LG A ⊢ B → LG D ⊖ A ⊢ C ⊖ B
  m⊘  : LG A ⊢ B → LG C ⊢ D → LG A ⊘ D ⊢ B ⊘ C
    -- grishin distributives
  d⊖↗ : LG A ⊗ B ⊢ C ⊕ D → LG C ⊖ A ⊢ D ↗ B
  d⊖↖ : LG A ⊗ B ⊢ C ⊕ D → LG C ⊖ B ⊢ A ↖ D
  d⊘↖ : LG A ⊗ B ⊢ C ⊕ D → LG B ⊘ D ⊢ A ↖ C
  d⊘↗ : LG A ⊗ B ⊢ C ⊕ D → LG A ⊘ D ⊢ C ↗ B
```

Note that Agda allows arbitrary unicode characters in identifiers, so r⊗↖ is a valid Agda identifier.

Using this data type, we can already do quite a lot. For instance, we can show that while the inference rule ax above is restricted to atomic formulas[3], the unrestricted version is admissible, by induction on the formula. Note that the construction {A = ...} below is used to pattern match on the implicit variable A:

[2] For the typeset version of this paper we omit the quantifiers for all implicit, universally quantified arguments.

[3] Whereas the rule ax may appear to be unrestricted, it only allows the derivation of the identity proof for any formula el A. That is, any *atomic formula* A delimited by the constructor el.

```
ax′ : LG A ⊢ A
ax′ {A = el   _} = ax
ax′ {A = _ ⊗ _} = m⊗ ax′ ax′
ax′ {A = _ ↙ _} = m↙ ax′ ax′
ax′ {A = _ ↖ _} = m↖ ax′ ax′
ax′ {A = _ ⊕ _} = m⊕ ax′ ax′
ax′ {A = _ ⊘ _} = m⊘ ax′ ax′
ax′ {A = _ ⊖ _} = m⊖ ax′ ax′
```

Alternatively, we could derive the various applications and co-applications that hold in the Lambek-Grishin calculus:

```
appl-↖′ : LG A ⊗ (A ↖ B) ⊢ B
appl-↖′ = r↖⊗ (m↖ ax′ ax′)
appl-↙′ : LG (B ↙ A) ⊗ A ⊢ B
appl-↙′ = r↙⊗ (m↙ ax′ ax′)
appl-⊖′ : LG B ⊢ A ⊕ (A ⊖ B)
appl-⊖′ = r⊖⊕ (m⊖ ax′ ax′)
appl-⊘′ : LG B ⊢ (B ⊘ A) ⊕ A
appl-⊘′ = r⊘⊕ (m⊘ ax′ ax′)
```

However, the most compelling reason to use the axiomatisation we have chosen, using residuation and monotonicity rules, is that cut becomes an admissible rule.

3 Admissible Cut

We would like to show that cut′ of type $LG\ \mathsf{A} \vdash \mathsf{B} \to LG\ \mathsf{B} \vdash \mathsf{C} \to LG\ \mathsf{A} \vdash \mathsf{C}$ is an admissible rule. The method of **?**, for the basic non-associative Lambek calculus, can be readily generalized to the case of LG:

(i) every connective is introduced *symmetrically* by a monotonicity rule or as an axiom;
(ii) every connective has one side (antecedent or succedent) where, if it occurs there at the top level, it cannot be taken apart or changed by any inference rule;
(iii) as a consequence of (ii), every formula has one side where, if it occurs there at the top level, it is immutable, i.e. there is no rule which can eliminate it;
(iv) due to (i) and (iii), when we find such an immutable formula, we can be sure that, stepping through the derivation, after some number of steps we will find the monotonicity rule which introduced that formula;
(v) due to the type of cut′, when we match on the cut formula B we will always have an immutable variant of that formula in either the first or the second argument of cut′;
(vi) finally, for each main connective there exists a rewrite rule which makes use of the facts in (iv) and (v) to rewrite an application of cut′: to two applications of cut′ on the arguments of the monotonicity rule which introduced the connective, chained together by applications of residuation (for binary connectives) or simply to a derivation (for atomic formulas). As an example, the rewrite rule for _⊗_ can be found in figure 1.

Fig. 1. Rewrite rule for cut on formula B ⊗ C.

We can model the view on the left-hand side of the rewrite rule in figure 1 as a data type. In order to construct this view for some suitable derivation f, we need two derivations, h_1 and h_2 and a derivation f′, which represents the arbitrary derivation steps taking (m⊗ h_1 h_2) back to f. Lastly, we include a proof pr of the fact that the reconstructed derivation f′ (m⊗ h_1 h_2) is identical to f:

```
data Origin (f : LG A ⊢ B ⊗ C) : Set where
  origin : (h₁ : LG E ⊢ B)
           (h₂ : LG F ⊢ C)
           (f′ : ∀ {G} → LG E ⊗ F ⊢ G → LG A ⊢ G)
           (pr : f ≡ f′ (m⊗ h₁ h₂))
             → Origin f
```

In the above snippet, we have chosen to leave the quantifier $\forall \{G\}$ explicit to stress that f' should work for *any* formula G, not only for $B \otimes C$.

All that remains now is to show that for any f of type $LG\ A \vdash B \otimes C$, we can construct such a view. We will attempt to do this by induction on the given derivation. Note that $\{\ \}0$ is the Agda syntax for a proof obligation. For clarity, I have added the types of the various subproofs f in comments:

```
find : (f : LG A ⊢ B ⊗ C) → Origin f
find (m⊗  f g) = origin f g id refl
find (r↖⊗ f)   = { }0  -- f : LG A₂ ⊢ A₁ ↖ B ⊗ C
find (r↗⊗ f)   = { }1  -- f : LG A₁ ⊢ B ⊗ C ↗ A₂
find (r⊕⊘ f)   = { }2  -- f : LG A₂ ⊢ A₁ ⊕ B ⊗ C
find (r⊕⊘ f)   = { }3  -- f : LG A₁ ⊢ B ⊗ C ⊕ A₂
```

Alas! While in the first case, where f is of the form $m\otimes\ f\ g$, we have found our monotonicity rule, the remaining cases are less kind. It seems that we have neglected to account for derivations where our cut formula is temporarily nested within another formula.

We will need some new vocabulary to describe what is going on in the above example. We would like to describe contexts which a) can be taken apart using residuation, and b) when fully taken apart, will leave the nested formula on the correct side of the turnstile. A natural fit for this is using polarity:

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

Below we define well-polarised formula and judgement contexts with exactly one hole. We use a $\triangleleft$ or $\triangleright$ to denote in which argument the hole is:

```
data Context (p : Polarity) : Polarity → Set where
  []       : Context p p
  _⊗▹_     : Type → Context p +  → Context p +
  _↖▹_     : Type → Context p −  → Context p −
  _↗▹_     : Type → Context p +  → Context p −
  _◃⊗_     : Context p +  → Type → Context p +
  _◃↖_     : Context p +  → Type → Context p −
  _◃↗_     : Context p −  → Type → Context p −
  _⊕▹_     : Type → Context p −  → Context p −
  _⊘▹_     : Type → Context p −  → Context p +
  _⦸▹_     : Type → Context p +  → Context p +
  _◃⊕_     : Context p −  → Type → Context p −
  _◃⊘_     : Context p +  → Type → Context p +
  _◃⦸_     : Context p −  → Type → Context p +
data Context^J (p : Polarity) : Set where
  _◃⊢_     : Context p +  → Type → Context^J p
  _⊢▹_     : Type → Context p −  → Context^J p
```

We also define two operators which, given a context and a formula, will fill the hole in the given context with the given formula. The definition for $\_\,[\_]$ is entirely predictable and repetitive, and has been mostly omitted[4]:

```
_[_] : Context p₁ p₂ → Type → Type
[]          [A] = A
(B ⊗▹ C) [A] = B ⊗ (C [A])
...
```

[4] For the remainder of this paper, any partial omission of a function will be denoted with an ellipsis at the end of the code block.

```
_ [_] ^J : Context^J p → Type → Judgement
(A ◃ ⊢ B) [C] ^J = A [C] ⊢ B
(A ⊢ ▹ B) [C] ^J = A ⊢ B [C]
```

The crucial point about these well-polarised judgement contexts is that, once the entire context is peeled away, the formula will be at the top level on the side corresponding to the polarity argument—with + and − corresponding to the antecedent and the succedent, respectively. Therefore, in order to generalise our previous definition of Origin, we want the occurrence of B ⊗ C to be nested in a *negative* context:

```
data Origin′ (J : Context^J −)
     (f  : LG J [B ⊗ C] ^J)
         : Set where
  origin : (h₁ : LG E ⊢ B)
     (h₂ : LG F ⊢ C)
     (f′ : LG E ⊗ F ⊢ G → LG J [G] ^J)
     (pr : f ≡ f′ (m⊗ h₁ h₂))
         → Origin′ J f
```

Using this more general definition Origin′, we can define a more general function find′—and this time, our proof by induction works!

Note that in Agda, the **with** construct is used to pattern match on the result of an expression:

```
find′ : (J : Context^J −) (f : LG J [B ⊗ C] ^J) → Origin′ J f
find′ (._ ⊢ ▹ [])          (m⊗ f g) = origin f g id refl
find′ (._ ⊢ ▹ (A ◃ ⁄ _)) (r⊗⁄ f)  with find′ (_ ⊢ ▹ A) f
... | origin h₁ h₂ f′ pr rewrite pr  = origin h₁ h₂ (r⊗⁄ ∘ f′) refl
find′ (._ ⊢ ▹ (_ ⧵ ▹ B)) (r⊗⧵ f)  with find′ (_ ⊢ ▹ B) f
... | origin h₁ h₂ f′ pr rewrite pr  = origin h₁ h₂ (r⊗⧵ ∘ f′) refl
...
```

However, there are many cases—53 in total. The reason for this is that the possible derivation steps depend on the main connective; therefore we first have to explore every possible main connective, and then every possible rule which would produce that main connective. Because of this, the definitions of the various find′ functions are very long and tedious, and have mostly been omitted.[5]

From the more general Origin′ and find′ we can very easily recover our original definitions Origin and find by setting the context to be empty. In the case of the cut formula B ⊗ C, we set the context to (_ ⊢ ▹ []) to ensure that the formula ends up at the top level in the succedent:

```
Origin : (f : LG A ⊢ B ⊗ C) → Set
Origin f = Origin′ (_ ⊢ ▹ []) f
find   : (f : LG A ⊢ B ⊗ C) → Origin f
find   f = find′   (_ ⊢ ▹ []) f
```

And with that, we can finally put the rewrite rules from **?** to use. We can define cut′ by pattern matching on the cut formula B; applying the appropriate find′ function to find′ the monotonicity rule introducing the formula; and apply the appropriate rewrite rule to create a derivation containing two cuts on structurally smaller formulas:

```
cut′ : (f : LG A ⊢ B) (g : LG B ⊢ C) → LG A ⊢ C
cut′ {B = el _}   f g with el.find g
... | (el.origin          g′ _) = g′ f
cut′ {B = _ ⊗ _} f g with ⊗.find f
... | (⊗.origin     h₁ h₂ f′ _) = f′ (r⁄⊗ (cut′ h₁ (r⊗⁄ (r⧵⊗ (cut′ h₂ (r⊗⧵ g))))))
cut′ {B = _ ⁄ _} f g with ⁄.find g
... | (⁄.origin     h₁ h₂ g′ _) = g′ (r⊗⁄ (r⧵⊗ (cut′ h₂ (r⊗⧵ (cut′ (r⁄⊗ f) h₁)))))
cut′ {B = _ ⧵ _} f g with ⧵.find g
... | (⧵.origin     h₁ h₂ g′ _) = g′ (r⊗⧵ (r⁄⊗ (cut′ h₁ (r⊗⁄ (cut′ (r⧵⊗ f) h₂)))))
cut′ {B = _ ⊕ _} f g with ⊕.find g
```

[5] The burden on the programmer or logician can be reduced by clever use of the symmetries $\cdot^{\bowtie}$ and $\cdot^{\infty}$ as done in **?**. One would have to implement only *three* of the find′ functions (e.g. for el, ⊗ and ⧵); the remaining four can then be derived using the symmetries.

```
... | (⊕.origin      h₁ h₂ g′ _) = g′ (r⊘⊕ (cut′ (r⊕⊘ (r⊘⊕ (cut′ (r⊕⊘ f) h₂))) h₁))
cut′ {B = _ ⊘ _} f g with ⊘.find f
... | (⊘.origin      h₁ h₂ f′ _) = f′ (r⊕⊘ (r⊘⊕ (cut′ (r⊕⊘ (cut′ h₁ (r⊘⊕ g))) h₂)))
cut′ {B = _ ⊘ _} f g with ⊘.find f
... | (⊘.origin      h₁ h₂ f′ _) = f′ (r⊕⊘ (r⊘⊕ (cut′ (r⊕⊘ (cut′ h₂ (r⊘⊕ g))) h₁)))
```

4 CPS Translation

For this paper, we have opted to implement the call-by-value CPS translation as described in **?**. This translation consists of three elements:

- a function $\lceil \_ \rceil$, which translates formulas in LG to formulas in the target system—while we have chosen to translate to Agda, the original translation targeted multiplicative intuitionistic linear logic;
- a pair of mutually recursive functions $\lceil \_ \rceil^L$ and $\lceil \_ \rceil^R$, which translate terms in LG to terms in the target system.

In order to write these functions, we will need two additional pieces of information: a function $\lceil \_ \rceil^A$, which translates the atomic formulas to Agda types; and a return type R, which we will use to define a "negation" as $\neg\ A\ =\ A \to R$. We will therefore implement the CPS translation in a sub-module, which abstracts over these terms:

```
module translation (⌈_⌉ᴬ : Atom → Set) (R : Set) where
```

When using this module, we will generally identify the return type R with the type Bool for booleans. However, abstracting over it will ensure that we do not accidentally use this knowledge during the translation.

The type-level translation itself maps formulas in LG to types in Agda, as follows:

```
⌈_⌉ : Type → Set
⌈ el   A ⌉ =       ⌈ A ⌉ᴬ
⌈ A ⊗ B ⌉ =    (   ⌈ A ⌉ ×   ⌈ B ⌉)
⌈ A ↖ B ⌉ = ¬ (   ⌈ A ⌉ × ¬ ⌈ B ⌉)
⌈ B ↗ A ⌉ = ¬ (¬ ⌈ B ⌉ ×   ⌈ A ⌉)
⌈ B ⊕ A ⌉ = ¬ (¬ ⌈ B ⌉ × ¬ ⌈ A ⌉)
⌈ B ⊘ A ⌉ =    (   ⌈ B ⌉ × ¬ ⌈ A ⌉)
⌈ A ⊘ B ⌉ =    (¬ ⌈ A ⌉ ×   ⌈ B ⌉)
```

The translations on terms map terms in LG to the Agda function space. Each LG term is associated with *two* functions, depending on whether the focus is on A or B as the active formula:

```
mutual
  ⌈_⌉ᴸ : LG A ⊢ B → ¬ ⌈ B ⌉ → ¬ ⌈ A ⌉
  ⌈_⌉ᴿ : LG A ⊢ B → ⌈ A ⌉ → ¬ ¬ ⌈ B ⌉
...
```

The CPS translations of the terms are rather verbose, and trivial to deduce, when guided by the translation on types. Therefore, in the interest of space they have been omitted from the paper.[6]

5 Example

In this final section, we will present the analysis of an example sentence, using the type-logical grammar implemented above. The example we will analyse is:

"Someone loves everyone."

This sentence is well known to be ambiguous, owing to the presence of the two quantifiers. There are two readings:

a. There is some person who loves every person.
b. For each person, there is some person who loves them.

[6] They are, however, present in the source and therefore available on GitHub.

We will demonstrate that the system, as implemented in this paper, accurately captures these readings.

Before we can do that, however, there is a small amount of boiler plate that we have to deal with: we still need to choose a representation for our atomic types, and show how these translate into Agda. In what follows, we will assume we have access to a type for entities, suitable definitions for the universal and existential quantifiers, and meanings for 'loves' and 'person':

```
postulate
  Entity : Set
  ∀      : (Entity → Bool) → Bool
  ∃      : (Entity → Bool) → Bool
  LOVES  : Entity → Entity → Bool
  PERSON : Entity → Bool
```

We will instantiate the type for atomic formulas to Atom, as defined below:

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

Last, we need to define a function which maps the values of Atom to Agda types. We would like to map the atomic formulas as follows:

```
⌈_⌉ᴬ : Atom → Set
⌈ N ⌉ᴬ = Entity → Bool
⌈ NP ⌉ᴬ = Entity
⌈ S ⌉ᴬ = Bool
```

Now that we have Atom and $\lceil \_ \rceil^A$, we can open up the modules defined as above, instantiating the return type R with the type of booleans.

```
open logic             Atom
open logic.translation Atom ⌈_⌉ᴬ Bool
```

With everything that we implemented in scope, we can now define a small lexicon for our example sentence.

In what follows, we will use the aliases n, np and s for el N, el NP and el S, respectively:

```
someone : ⌈ (np ↗ n) ⊗ n ⌉
someone = ((λ {(g , f) → ∃ (λ x → f x ∧ g x)}) , PERSON)
loves   : ⌈ (np ↖ s) ↗ np ⌉
loves   = λ {(k , y) → k (λ {(x , k) → k (LOVES x y)})}
everyone : ⌈ (np ↗ n) ⊗ n ⌉
everyone = ((λ {(g , f) → ∀ (λ x → f x ⊃ g x)}) , PERSON)
```

Given the types we used for our lexical entries, the judgement which asserts the grammaticality of our sentence becomes:

```
((np ↗ n) ⊗ n) ⊗ (((np ↖ s) ↗ np) ⊗ ((np ↗ n) ⊗ n)) ⊢ s
```

There are seven proofs of this judgement. Below we have included the first *two* proofs:[7]:

```
SENT₀ = r↖⊗ (r↗⊗ (m↗ (m↖ (r↗⊗ ax′) ax) (r↗⊗ ax′)))
SENT₁ = r↗⊗ (r↗⊗ (m↗ (r⊗↗ (r↖⊗ (r↗⊗ (m↗ ax′ (r↗⊗ ax′))))) ax))
...
```

We can now apply our CPS translation to compute the denotations of our sentence, passing in the denotations of the words as a tuple, and passing in the identity function as the last argument in order to obtain the result:

```
sent₀ : ⌈ SENT₀ ⌉ᴿ (someone , loves , everyone) id ↦ ∀ (λ y → PERSON y ⊃ ∃ (λ x → PERSON x ∧ LOVES x y))
sent₁ : ⌈ SENT₁ ⌉ᴿ (someone , loves , everyone) id ↦ ∃ (λ x → PERSON x ∧ ∀ (λ y → PERSON y ⊃ LOVES x y))
...
```

Voila! Our system produces exactly the expected readings.

[7] We have chosen not to include the other five proofs as, under the CPS translation, they have the same interpretations as either the first or the second proof. For the interested reader, however, the proofs are present in the source, and therefore available on GitHub.

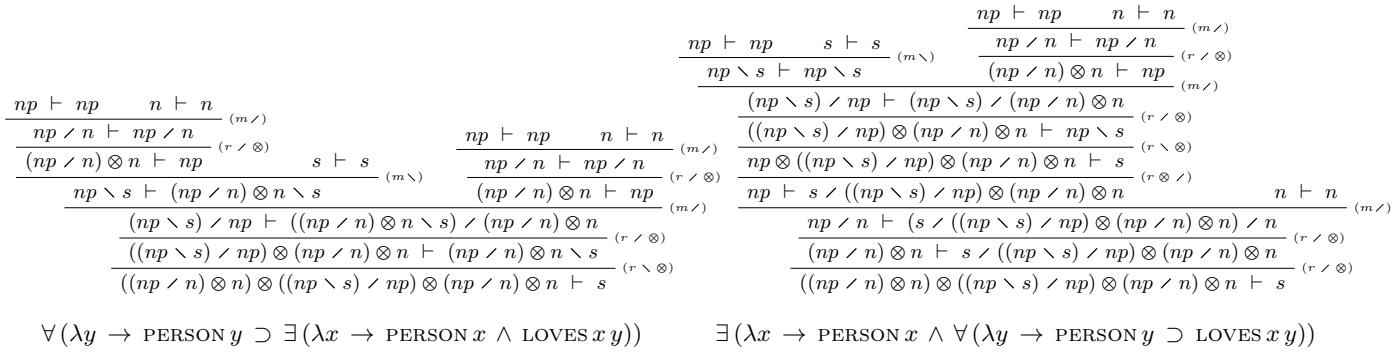

Fig. 2. "Someone loves everyone."

6 Conclusion

We have presented the reader with a simple formalisation of the Lambek-Grishin calculus, using the proof assistant Agda. We have shown how to formalise the proof of the admissibility of cut from **?** in Agda, and have extended this proof to cover all of LG. While we have not covered any of the usual unary operators, the formalism presented here generalises straightforwardly to accommodate connectives of any arity—and this extension, together with many other extensions, are indeed implemented in the full version of our code.

We have then presented the reader with a call-by-value CPS translation into the host language Agda, and used this translation to demonstrate the analysis of an example sentence.

Most importantly, we hope we presented the reader with a clean and readable formalisation of the Lambek-Grishin calculus.

7 Related & Future Work

Previous work on the formalisation of Lambek calculi was done in Coq by **?**, who, amongst other things, implemented sequent calculus and natural deduction systems for multi-modal categorial grammars.

The work presented in this paper is part of a larger undertaking to formalise type-logical grammars in Agda. At the moment, we have formalised not only the algebraic Lambek-Grishin calculus—which was presented in this paper—but also structural and polarised varieties of this calculus. From these implementations, we are able to extract implementations of their respective non-associative Lambek calculi.

In addition, we have implemented various other multi-modal systems, such as NL_{CL} (**?**).

We aim to extend this work by further formalising the known work w.r.t. these calculi, and creating tools to accommodate the writing of formal linguistics papers in literate style.

References

Anoun, H. (2007). Une bibliothèque coq pour le traitement des langues naturelles. *Technique et Science Informatiques*, 26(9):1111–1136.

Anoun, H., Castéran, P., and Moot, R. (2004). Proof automation for type-logical grammars. Rapport de recherche , European Summer School in Logic, Language and Information - 2004.

Asher, N. (2011). *Lexical Meaning in Context: A Web of Words*. Cambridge University Press.

Barker, C. and Shan, C. (2015). *Continuations and Natural Language (Oxford Studies in Theoretical Linguistics)*. Oxford University Press.

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

Moortgat, M. (2007). Symmetries in natural language syntax and semantics: The lambek-grishin calculus. In *Logic, Language, Information and Computation*, pages 264–284. Springer Berlin Heidelberg.

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

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*. Springer Berlin Heidelberg.

Norell, U. (2009). Dependently typed programming in agda. In *Proceedings of the 4th International Workshop on Types in Language Design and Implementation*, TLDI '09, pages 1–2, New York, NY, USA. ACM.