# Symmetric Categorial Grammar in Agda

Pepijn Kokke

March 27, 2015

## Abstract

In recent years, the interest in using proof assistants to reason about categorial grammars has grown.

## 1 Introduction

## 2 Types, Judgements, Base System

If we want to model our categorial grammars in Agda, a natural starting point would be our atomic types—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 types, and side-step the debate on which types *should* be atomic, we will simply assume there is a some data type representing our atomic types.

```
module main (Univ : Set) where
```

Our types can easily be described as a data type, injecting our atomic types by means of the constructor el, and adding the familiar connectives from the Lambek Grishin calculus as binary constructors.[1] In the same manner, we will define a data type to represent judgements.

```
data Type : Set ℓ where
  el                  : Univ → Type
  _⊗_ _↖_ _↗_ : Type → Type → Type
  _⊕_ _⊘_ _⦸_ : Type → Type → Type
data Judgement : Set ℓ where
  _⊢_ : Type → Type → Judgement
```

[1] Agda uses underscores in definitions to denote the argument positions, so _⊗_ defines an infix, binary connective.

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 our dependent type system gets a chance to shine! The constructors for our data type will represent our axiomatic rules, 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 ℓ 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
```

Using this data type we can already do quite a lot. For instance, we can show that while ax above is restricted to atomic types, the unrestricted version is

[2] For the typeset version of this paper we omit all implicit, universally quantified arguments, as is conventional in both Haskell (**?**) and Idris (**?**), and in much of logic.

admissible, by induction on the type.

```
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′
```

Or we could derive the various (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′)
```

But neither of those is really interesting compared to what is probably one of the most compelling reasons to use this axiomatisation...

# 3 Admissible Transitivity

We would like to show that trans′ 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 conventional proof for this reads as follows:

(i) every connective is introduced *symmetrically* by a monotonicity rule or axiom;

(ii) every connective has a side (antecedent or succedent) where, if it occurs there at the top level, it cannot be taken apart or moved by any inference rule;

(iii) due to (i) and (ii), when we find such an *immovable* connective, we can be sure that after an arbitrary number of proof steps we will find the monotonicity rule which introduces that connective;

(iv) due to the type of trans′, when we match on the cut formula B, regardless of the main connective in B, we will always have a proof with an immovable variant of that connective;

(v) finally, for each connective there exists a rewrite schema which makes use of the facts in (iii) and (iv) to rewrite an application of trans′ to two smaller applications of trans′ on the arguments of the connective (for binary connectives), or simply to a proof (in the case of atomic formulas). For example, the rewrite schema for _⊗_ can be found in figure 1.

### Formula Contexts

```
data Context : Set ℓ where
  []                    : Context
  _ ⊗›_ _ ⟍›_ _ ⟋›_     : Type    → Context → Context
  _ ⊕›_ _ ⦸›_ _ ⦸›_     : Type    → Context → Context
  _‹⊗ _ _‹⟍ _ _‹⟋ _     : Context → Type    → Context
  _‹⊕ _ _‹⦸ _ _‹⦸ _     : Context → Type    → Context

_[_]  : Context → Type    → Type
_⟨_⟩  : Context → Context → Context

data Polarised (p : Polarity) : Polarity → Context → Set ℓ where
  []      : Polarised p p []
  _ ⊗›_ : (A : Type) {B : Context}
        → Polarised p + B
        → Polarised p + (A ⊗› B)
  _ ⟍›_ : (A : Type) {B : Context}
        → Polarised p − B
        → Polarised p − (A ⟍› B)
  ...
```

### Origins

```
data Origin (J⁻ : Polarised − J) (f : LG J [B ⊗ C] ) : Set ℓ where
  origin : (h₁ : LG E ⊢ B)
           (h₂ : LG F ⊢ C)
           (f′  : ∀ {G} → LG E ⊗ F ⊢ G ⋯ J [G] )
           (pr  : f ≡ f′ $ m⊗ h₁ h₂)
               → Origin J⁻ f

viewOrigin : (J⁻ : Polarised − J) (f : LG J [B ⊗ C] ) → Origin J⁻ f
viewOrigin (._ ⊢› []) (m⊗ f g) = origin f g [] refl
...
```

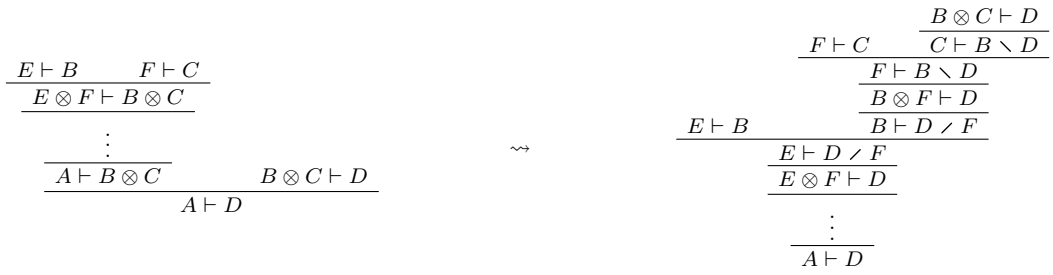

Figure 1: Rewrite schema for cut on formula $\mathsf{B} \otimes \mathsf{C}$.

```
trans′ : ∀ {A B C} (f : LG A ⊢ B) (g : LG B ⊢ C) → LG A ⊢ C
trans′ {B = el _}    f g with el.viewOrigin ([] ‹ ⊢ _) g
... | (el.origin          g′ pr) = g′ f
trans′ {B = _ ⊗ _} f g with ⊗.viewOrigin (_ ⊢ › []) f
... | (⊗.origin h₁ h₂ f′ pr)  = f′ (r⁄⊗ (trans′ h₁ (r⊗⁄ (r⧵⊗ (trans′ h₂ (r⊗⧵ g))))))
trans′ {B = _ ⁄ _} f g with ⁄.viewOrigin ([] ‹ ⊢ _) g
... | (⁄.origin h₁ h₂ g′ pr)  = g′ (r⊗⁄ (r⧵⊗ (trans′ h₂ (r⊗⧵ (trans′ (r⁄⊗ f) h₁)))))
trans′ {B = _ ⧵ _} f g with ⧵.viewOrigin ([] ‹ ⊢ _) g
... | (⧵.origin h₁ h₂ g′ pr)  = g′ (r⊗⧵ (r⁄⊗ (trans′ h₁ (r⊗⁄ (trans′ (r⧵⊗ f) h₂)))))
trans′ {B = _ ⊕ _} f g with ⊕.viewOrigin ([] ‹ ⊢ _) g
... | (⊕.origin h₁ h₂ g′ pr)  = g′ (r∅⊕ (trans′ (r⊕∅ (r⦸⊕ (trans′ (r⊕⦸ f) h₂))) h₁))
trans′ {B = _ ∅ _} f g with ∅.viewOrigin (_ ⊢ › []) f
... | (∅.origin h₁ h₂ f′ pr)  = f′ (r⊕∅ (r⦸⊕ (trans′ (r⊕⦸ (trans′ h₁ (r∅⊕ g))) h₂)))
trans′ {B = _ ⦸ _} f g with ⦸.viewOrigin (_ ⊢ › []) f
... | (⦸.origin h₁ h₂ f′ pr)  = f′ (r⊕⦸ (r∅⊕ (trans′ (r⊕∅ (trans′ h₂ (r⦸⊕ g))) h₁)))
```

Figure 2: Proof of admissible transitivity.