Modelling Substructural Logics in Agda

Pepijn Kokke

February 18, 2014

Abstract

In this paper, we will examine models of substructural logics in Agda. The reason for this is that most existing models formalise intuitionistic logic and are entirely unsuitable to modelling substructural logics. In recent years, however, substructural logics have seen a surge in usage.

Concretely we present the reader with an explicit model of intuitionistic logic, and derive models for linear logic and the Lambek-Grishin calculus. In addition, we show how to reify proofs in these logics into terms in Agda. All this is implemented as an Agda library, which is made available on GitHub.

Finally we conclude with an example from formal linguistics in which we demonstrate one possible usage of our implemented Agda library.

1 Introduction

You can find implementations of the simply-typed lambda calculus in Agda all across the web—for instance, the implementations by Mazzoli (2013), Érdi (2013) or Mu (2008). It is used as a running example in Norell's *Introduction to Agda*, and Érdi goes as far as to call it the "FizzBuzz of dependently-typed programming"—the problem that any capable programmer in the field should be able to solve.

Though each of these implementation has its own merits, they are all loosely based on the following model of the simply-typed lambda calculus.[1]

```
data _⊢_ : {k : ℕ} (Γ : Vec Type k) (A : Type) → Set where
  var : (x : Fin k) → Γ ⊢ lookup x Γ
  abs : A, Γ ⊢ B → Γ ⊢ A ⇒ B
  app : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
```

The advantages of using such a model are plenty. For instance, you can use Agda's built-in type-checker to verify the correctness of your proofs; and you can use the interactive proof assistant to develop your proofs.

This paper has three main contributions; we will present

[1]It should be noted that for the sake of readability in this paper implicit arguments are often left out. Any undefined variable that is encountered upon reading should be considered implicitly quantified over unless noted otherwise.

- an investigation into the modelling of logics in Agda;
- an investigation into the modelling of *substructural* logics in Agda;
- and—concretely—models of linear logic and of the Lambek-Grishin calculus, and a verification of the correctness of their interpretations in intuitionistic logic.

Below we will briefly motivate these contributions separately.

Why model logics in Agda at all? Why should we attempt to model logics at all? In our opinion there are several good reasons for doing this.

First of all, creating a formal model of a logical system forces you to make every detail of the system explicit. Not only may this help you by revealing small errors that would otherwise have gone unnoticed, but it also forces you to scrutinise the precise formulation of your axioms.[2]

Secondly, a model of a logical system in Agda is more than just a proof of its sanity. It is also a direct implementation of the calculus, which allows you play with your logic in a computational environment, using inference rules and proofs as first-class citizens. In addition to this, as mentioned before, the correctness of your proofs is checked by Agda's type-checker; and you can use theorem provers built in or for Agda, such as Agsy (Lindblad and Benke, 2006), to prove theorems in your modelled logic.

Lastly, for logics which have a computational interpretation in intuitionistic logic, you can translate proofs in the modelled logic to terms in Agda, which allows you to use Agda's built-in mechanisms for reduction and evaluation.

Why should we model *substructural* logics in Agda? As discussed above, most models of logic currently implemented in Agda formalise intuitionistic logic. In addition, the manner in which these models are implemented usually leaves the structural rules implicit, making them unsuitable for formalising substructural rules.

In recent years, however, substructural logics have seen a surge in fields as diverse as philosophy (relevant logics), linguistics (the Lambek calculus) and computing science (linear logic) (Restall, 2011). We therefore think it useful to examine the modelling of such logics in Agda as well.

Furthermore, when viewed from the perspective of Agda, if we can formalise a logic with certain properties (such as linearity for linear logic), then we can easily prove that, when we reify terms of this logic back into Agda, the corresponding Agda terms will share this property. This allows us to embed, for instance, provably linear lambda terms in Agda.

[2] An example: a common formulation of the exchange principle is $\Gamma, B, A, \Delta \vdash C \to \Gamma A, B, \Delta \vdash C$. However, using this principle to define, for instance, the swapping of two contexts $\Delta, \Gamma \vdash A \to \Gamma, \Delta \vdash A$ requires a number of applications quadratic in the lengths of Γ and Δ.

Why model the Lambek-Grishin calculus? The formulation of the Lambek-Grishin calculus (**LG**) modelled in this paper is quite a complex system. It is a substructural logic based on the non-associative Lambek calculus (**NL**), but adds the dual for each connective (Moortgat and Moot, 2013). It is formulated in the style of display logic (Belnap, 1982), and uses techniques such as polarisation and focusing (Andreoli, 1992). We therefore feel that it would be an interesting enterprise to model the Lambek-Grishin calculus, as it allows us to examine not only the formalisation of substructure in isolation, but also in the presence of other techniques.

And, since **LG** is a very complex logical system, we hope that an explicit and interactive formalisation may be able to aid students in understanding it—especially those coming from a background in computer science.

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 Norell (2009) for a detailed discussion of Agda in general, or to the list of Agda tutorials maintained on the Agda website.[3]

Before we start off, 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.[4]

2 Intuitionistic Logic

2.1 Modelling IL with de Bruijn indices

If we wish to model the intuitionistic calculus, we first have to do something about our notation. The reason for this is that the usual notation with named variables introduces a whole host of problems, such as checking for proper scopal and binding relations.[5]

The canonical solution to this is a notation introduced in de Bruijn (1972), where we instead of using variable names for binding, we will use numbers. The semantics of these numbers will be that they tell you how many lambdas up the variable is bound (or, from the perspective of logic, they are indices into the context). See Figure 1 for an example of how terms in named notation compare to terms in de Bruijn notation.

As a preparation for the modelling of the intuitionistic calculus in Agda, we can formulate the de Bruijn notation as a set of inference rules; the result of this can be seen in Figure 2.

As a first step of we will need a representation of the type language/formulas that we wish to model. In this paper we will limit ourselves to formulas containing implication (written $\_ \Rightarrow \_$) and conjunction (written $\_ \times \_$).

[3] See `http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Othertutorials`.
[4] See `https://github.com/pepijnkokke/SubstructuralLogicsInAgda`.
[5] See Érdi (2013) for an implementation that uses variable names.

Named	de Bruijn
$\lambda x \to x$	$\lambda\ 0$
$\lambda x \to \lambda y \to x$	$\lambda\ \lambda\ 1$
$\lambda x \to \lambda y \to \lambda z \to x\ z\ (y\ z)$	$\lambda\ \lambda\ \lambda\ 2\ 0\ (1\ 0)$

Figure 1: Named notation versus de Bruijn notation (Mazzoli, 2013).

$$\frac{}{\Gamma \vdash (var\ i) : \Gamma_i}\ \textsc{ax}$$

$$\frac{\Gamma, A \vdash t : B}{\Gamma \vdash (abs\ t) : A \Rightarrow B}\ \Rightarrow\text{-intro} \qquad \frac{\Gamma \vdash s : A \Rightarrow B \qquad \Gamma \vdash t : A}{\Gamma \vdash (app\ s\ t) : B}\ \Rightarrow\text{-elim}$$

$$\frac{\Gamma \vdash s : A \qquad \Gamma \vdash t : B}{\Gamma \vdash (pair\ s\ t) : A \times B}\ \times\text{-intro} \qquad \frac{\Gamma \vdash s : A \times B \qquad A, B, \Gamma \vdash t : C}{\Gamma \vdash (case\ s\ t) : C}\ \times\text{-elim}$$

Figure 2: Inference rules for **IL** corresponding to the de Bruijn notation.

In addition, we will abstract over some type U. The reason for this is that we do not want to be forced to specify the atomic types—instead we shall allow the user to provide their own universe of atomic types.[6]

```
data Type : Set where
  el    : (A : U) → Type
  _×_   : Type → Type → Type
  _⇒_   : Type → Type → Type
```

All that is left for us to do, is to translate our inference rules as presented in Figure 2 to an Agda data type. The translation is almost verbatim, save that we write _ → _ for the meta-logical implication (instead of a horizontal line) and—due to the close relationship between proofs and terms—the term constructors (var, abs, case, et cetera) become constructors of our data type.

We use vectors[7] to model contexts, and finite sets[8] to model the de Bruijn indices. In this way we can ensure that every variable is bound,[9] either to a type in the context or to a lambda abstraction.[10] Because of this invariant we can define a safe lookup function as follows.

```
lookup : Fin k → Vec A k → A
lookup zero    (A, Γ)  =  A
lookup (suc x) (A, Γ)  =  lookup x Γ
```

[6] For an example of this, see §§ 4.5.

[7] See http://agda.github.io/agda-stdlib/html/Data.Vec.html#604.

[8] See http://agda.github.io/agda-stdlib/html/Data.Fin.html#775.

[9] The reason this works is because vectors encode lists of a fixed length k, and finite sets encode a data type with precisely k inhabitants.

[10] It should be stated that throughout this paper we will use an alternative notation for lists and vectors, using _, _ for the cons operator and $\emptyset$ for the empty list (or vector), as we deem this notation to be better suited to sequent contexts. For the concatenation of contexts, however, we will stick to using _ ++ _, as usual.

And using this function we can present a full formalisation of our inference rules.

```
data _⊢_ : (Γ : Vec Type k) (A : Type) → Set where
  var  : (x : Fin k) → Γ ⊢ lookup x Γ
  abs  : A, Γ ⊢ B → Γ ⊢ A ⇒ B
  app  : Γ ⊢ A ⇒ B → Γ ⊢ A → Γ ⊢ B
  pair : Γ ⊢ A → Γ ⊢ B → Γ ⊢ A × B
  case : Γ ⊢ A × B → A, B, Γ ⊢ C → Γ ⊢ C
```

In § 1 we mentioned that one of the advantages of modelling a logic in Agda was the use of Agda's interactive proof assistant. Below we will demonstrate how the proof assistant might be used to formulate a proof.

Agda allows you to leave holes in expressions; for every hole you leave, Agda will report the type of the expressions needed to plug that hole. For instance, in the example below we try to prove the commutativity of _×_. After an initial lambda abstraction, we leave a hole, and Agda tells us the type it is expecting at that position.

```
swap : Γ ⊢ A × B ⇒ B × A
swap = abs { }0

{ }0 : A × B, Γ ⊢ B × A
```

This provides us with enough information to continue the proof. Let us assume that after introducing a case statement and a pair introduction, we become confused about the exact order our variables are in, so we once again ask Agda to tell us which sub-proofs we need to give.

```
swap : Γ ⊢ A × B ⇒ B × A
swap = abs (case (var zero) pair ({ }0 { }1))

{ }0 : A, B, A × B, Γ ⊢ B
{ }1 : A, B, A × B, Γ ⊢ A
```

This gives us enough information to complete the proof entirely—in fact, both holes can be trivially filled by the Agsy theorem prover,[11] resulting in the following proof.

```
swap : Γ ⊢ A × B ⇒ B × A
swap = abs (case (var zero) (pair (var (suc zero)) (var zero)))
```

This proof corresponds to the following typeset proof in natural deduction style.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{A, B, \Gamma \vdash (var\ 1) : B}\,\text{AX} \qquad \dfrac{}{A, B, \Gamma \vdash (var\ 0) : A}\,\text{AX}}{A, B, \Gamma \vdash (pair\ (var\ 1)\ (var\ 0)) : B \times A}\,\times\text{-intro} \qquad \dfrac{}{A \times B, \Gamma \vdash (var\ 0) : A \times B}\,\text{AX}}{A \times B, \Gamma \vdash (case\ (var\ 0)\ (pair\ (var\ 1)\ (var\ 0))) : B \times A}\,\times\text{-elim}}{\Gamma \vdash abs\ (case\ (var\ 0)\ (pair\ (var\ 1)\ (var\ 0))) : A \times B \Rightarrow B \times A}\,\Rightarrow\text{-intro}$$

[11] See http://wiki.portal.chalmers.se/agda/pmwiki.php?n=Main.Auto.

2.2 Exchange as admissible rule

While the above model of **IL** suffices if all we wish to model is the intuitionistic calculus, it poses some problems if we wish to model substructural logics such as linear logic.

The reason for this is that the structural rules (exchange, weakening and contraction) are admissible rules in our formulation, i.e. they are implicitly present in our formulation of **IL**.

We shall demonstrate this by giving an explicit formulation of the following simple exchange principle: exchange at the i-th position.

$$\frac{\Gamma, B, A, \Delta \vdash C}{\Gamma, A, B, \Delta \vdash C}$$

We can define this principle in three steps. First we define what exchange does on the level of contexts.

```
exch : (i : Fin k) → Vec Type (suc k) → Vec Type (suc k)
exch zero    (A, B, Γ) = B, A, Γ
exch (suc i) (A, Γ)    = A, (exch i Γ)
```

Secondly, we define what an exchange does at the level of the indices. Note that the way we implement this is by returning a new index, paired with a proof that the lookup function will return the same result when we use this new index with the exchanged context, as when we use the old index with the old context.

```
lemma-var : ∀ i x → ∃ λ y → lookup x Γ ≡ lookup y (exch i Γ)
lemma-var zero    zero          = suc zero, refl
lemma-var zero    (suc zero)    = zero, refl
lemma-var zero    (suc (suc x)) = suc (suc x), refl
lemma-var (suc i) zero          = zero, refl
lemma-var (suc i) (suc x)       = map suc id (lemma-var i x)
```

Finally, we can define exchange as a recursive function over proofs, where we recursively apply exchange to every sub-proof until we reach the axioms (or variables), at which point we use the lemma we defined above.[12]

```
exch : ∀ i → Γ ⊢ A → exch i Γ ⊢ A
exch i (var x) with lemma-var i x
exch i (var x) | y, p rewrite p = var y
exch i (abs t)      = abs (exch (suc i) t)
exch i (app s t)    = app (exch i s) (exch i t)
exch i (pair s t)   = pair (exch i s) (exch i t)
exch i (case s t)   = case (exch i s) (exch (suc (suc i)) t)
```

[12] Note that this is also what makes exchange admissible instead of derivable: we need to inspect the proof terms in order to be able to define it.

2.3 Explicit structural rules

If we wish to make our model of **IL** suitable for modelling substructural logics, we will have to remove the implicit exchange, weakening and contraction from our axioms, and add them as axioms in their own right.

The reason that the structural rules are implicitly present in our logic, is that all premises in our inference rules share a context. If we make sure that every premise of a rule has its own context, and all contexts are concatenated in the conclusion, we will have solved our issue. A surprising side-effect of this is that variables will no longer be needed—simply marking the position in a term as an axiom is sufficient. As a consequence of this, we can stop using vectors for modelling contexts, and switch to using simple lists.

Below you will find a model of **IL** in which the structural rules have been made explicit.[13]

```
data _⊢_ : ∀ (X : List Type) (A : Type) → Set where
  var  : A, ∅ ⊢ A
  abs  : A, X ⊢ B → X ⊢ A ⇒ B
  app  : X ⊢ A ⇒ B → Y ⊢ A → X ++ Y ⊢ B
  pair : X ⊢ A → Y ⊢ B → X ++ Y ⊢ A × B
  case : X ⊢ A × B → A, B, Y ⊢ C → X ++ Y ⊢ C
  weak : X ⊢ A → X ++ Y ⊢ A
  cont : A, A, X ⊢ B → A, X ⊢ B
  exch :  (X ++ Z) ++ (Y ++ W) ⊢ A
       → (X ++ Y) ++ (Z ++ W) ⊢ A
```

Using this model we can once again prove our running example—the commutativity of _ × _.[14]

```
swap : X ⊢ A × B ⇒ B × A
swap = abs (case var (exch₀ (pair var (weak var))))
```

2.4 Reification into Agda

Another advantage of modelling logics we mentioned in § 1 is that one can reify proofs in the modelled system back into Agda terms, and in this way piggyback on Agda's evaluation mechanisms.

In this section we will present a reification of our explicit **IL** terms into Agda terms. But first we will give a general definition of what a reification is.

A reification generally consists of two parts:[15]

- a translation function (written ⟦_⟧) that sends types in the source logic to types in the target logic;

[13] Note that the exchange principle we chose here is slightly different from the exchange principle we proved above, in that it allows the exchange of entire contexts.

[14] In the proof we use a derived inference rule, exch$_0$, which has the type $B, A, X \vdash C \to A, B, X \vdash C$, i.e. it exchanges the first two types in the context.

[15] Implicit in the below definition are the data types for the types of the source and target logics, and the data types for the proofs or terms of the source and target logics.

◦ a translation function (written [_]) that sends terms in the source logic to terms in the target logic.

First of all, let us look at the translation of our **IL** types into Agda's Set. Since we cannot know what to map types in the user-provided universe U to, we shall require the user to provide us with a translation function $[\![\_]\!]^u$. With this function, the full translation is trivial.

```
⟦_⟧ : Type → Set
⟦ el A     ⟧ = ⟦ A ⟧ᵘ
⟦ A × B  ⟧ = ⟦ A ⟧ × ⟦ B ⟧
⟦ A ⇒ B ⟧ = ⟦ A ⟧ → ⟦ B ⟧
```

Next we will look at the translation of proofs into Agda terms. Unfortunately for us, Agda has no explicit notion of contexts. We could therefore require that the proofs we translate are closed terms, i.e. of the form $\emptyset \vdash A$. Another solution, however, is to invent our own encoding of contexts.

So let us ask ourselves, what is a context? The answer: a context is a list of types associated with a list of values of those types. Or to phrase it the other way around: a list of values typed by a list of types. This means we can use heterogeneous lists (Kiselyov et al., 2004) to encode our contexts.

```
data Ctxt : ∀ (X : List Set) → Set₁ where
  ∅ : Ctxt ∅
  _,_ : A → Ctxt X → Ctxt (A, X)
```

Using this definition, our representation of a sequent $X \vdash A$ in Agda will be the type $Ctxt\ [\![X]\!] \to [\![A]\!]$.

Next we need a few simple functions to work with these contexts. Specifically, an exch function, which applies our exchange principle to the context, and a split function, which splits a context into two parts (for binary rules).

```
exch : Ctxt ((X ++ Y) ++ (Z ++ W)) → Ctxt ((X ++ Z) ++ (Y ++ W))
split : Ctxt (X ++ Y) → (Ctxt X) × (Ctxt Y)
```

For brevity's sake, we will omit the definitions for these functions. The interested reader can refer to the code for a full account. Instead we will present the reader with the full reification into Agda.

The reification is fairly straightforward: we simply have to map the constructors of our model to the corresponding constructs in Agda.

In the case of *variables*, we know that the environment will contain exactly one value of exactly the right type, for *lambda abstractions*, we abstract over a value, which we insert into the context, et cetera. Note that for binary rules we have to split the context, and pass the two parts down the corresponding branches of the proof during reification.

```
reify : X ⊢ A → (Ctxt ⟦ X ⟧ → ⟦ A ⟧)
reify var       (x, ∅) = x
reify (abs t)   E      = λ x → reify t (x, E)
reify (app s t)  E       with split E
... | Eˢ, Eᵗ              = (reify s Eˢ) (reify t Eᵗ)
reify (pair s t)  E      with split E
... | Eˢ, Eᵗ              = (reify s Eˢ, reify t Eᵗ)
reify (case s t) E       with split E
... | Eˢ, Eᵗ              = case reify s Eˢ of λ {(x, y) → reify t (x, y, Eᵗ)}
reify (weak s)  E        with split E
... | Eˢ, Eᵗ              = reify s Eˢ
reify (cont t)   (x, E)  = reify t (x, x, E)
reify (exch t)   E       = reify t (exch E)
```

We can now define the reification function [_] as a simple alias.

```
[_] : X ⊢ A → (Ctxt ⟦ X ⟧ → ⟦ A ⟧)
[_] = reify
```

And translate our running example into Agda. Note that we can already insert the empty context, as our example is a closed term.

```
swap′ : ⟦ A ⟧ × ⟦ B ⟧ → ⟦ B ⟧ × ⟦ A ⟧
swap′ = [swap] ∅
```

3 Linear Logic

3.1 Moving down to linear logic

As we have taken care to make all structural rules explicit, moving down to intuitionistic linear logic (**LP**) from our current model of **IL** is trivial. As a first step we define a new model for our types, to match the conventions of linear logic (we are adding bottom as an atomic type here, as we will need it later on).

```
data Type : Set where
  el     : (A : U) → Type
  ⊥      : Type
  _⊗_    : Type → Type → Type
  _⊸_    : Type → Type → Type
```

Next we can create the model of **LP** by copying our explicit model for **IL**, and simply removing the axioms for weakening and contraction.

```
data _⊢_ : ∀ (X : List Type) (A : Type) → Set where
  var  : A, ∅ ⊢ A
  abs  : A, X ⊢ B → X ⊢ A ⊸ B
  app  : X ⊢ A ⊸ B → Y ⊢ A → X ++ Y ⊢ B
  pair : X ⊢ A → Y ⊢ B → X ++ Y ⊢ A ⊗ B
  case : X ⊢ A ⊗ B → A, B, Y ⊢ C → X ++ Y ⊢ C
  exch :   (X ++ Z) ++ (Y ++ W) ⊢ A
         → (X ++ Y) ++ (Z ++ W) ⊢ A
```

And, since we added an atomic type for bottom, we can also add the usual definition for negation.

```
¬_ : Type → Type
¬ A = A ⊸ ⊥
```

Now we can define our running example. In fact, the definition has hardly changed since § 2. The only difference is that now the term *has to be closed*, as swapping in the presence of a context is not linear.

```
swap : ∅ ⊢ A ⊗ B ⊸ B ⊗ A
swap = abs (case var (exch₀ (pair var var)))
```

As a new example, we can also give a proof for the validity of type-raising.

```
raise : X ⊢ A → X ⊢ (A ⊸ B) ⊸ B
raise t = abs (app var t)
```

3.2 Reification into IL

We could define the reification of **LP** into Agda as we showed for **IL**, but it is much easier to translate our proofs to **IL** and use the previously defined reification.

We first define a translation of our types into the types of **IL**. Note that we have abstracted over an element of the user-provided type universe U—called R—to which we will map bottom in the translation of our types.

```
⟦_⟧ : Type → Type^IL
⟦ ⊥       ⟧ = el R
⟦ el A    ⟧ = el A
⟦ A ⊗ B   ⟧ = ⟦ A ⟧ × ⟦ B ⟧
⟦ A ⊸ B ⟧ = ⟦ A ⟧ ⇒ ⟦ B ⟧
```

Next we define a translation function that maps contexts in **LP** to contexts **IL**. Note that the implementation simply applies the translation function to every element in the context.

```
⟦_⟧ : List Type → List Type^IL
⟦_⟧ = map ⟦_⟧
```

Last, we define a translation from **LP** to **IL**. The translation is almost able to reconstruct the proof in **IL** verbatim, though we are omitting some minor details.[16]

```
toIL : X ⊢ A → ⟦ X ⟧ ⊢_IL ⟦ A ⟧
toIL var        = var
```

[16] The problematic details have to do with the application of $[\![\_]\!]$ to contexts; we have to rewrite using a lemma that states that $[\![X +\!\!+ Y]\!] \equiv [\![X]\!] +\!\!+ [\![Y]\!]$, i.e. that our translation commutes over context concatenation, for every binary rule.

```
toIL (abs t)     = abs (toIL t)
toIL (app s t)   = app (toIL s) (toIL t)
toIL (pair s t)  = pair (toIL s) (toIL t)
toIL (case s t)  = case (toIL s) (toIL t)
toIL (exch t)    = exch (toIL t)
```

Then we can define the reification of closed terms into Agda by simple function composition.

```
[_] : X ⊢ A → (Ctxt ⟦ ⟦ X ⟧ ⟧ → ⟦ ⟦ A ⟧ ⟧)
[_] = [_]^IL ∘ toIL
```

And again, we can reify our (now linear) swap function back into Agda.

```
swap′ : ⟦ ⟦ A ⟧ ⟧ × ⟦ ⟦ B ⟧ ⟧ → ⟦ ⟦ B ⟧ ⟧ × ⟦ ⟦ A ⟧ ⟧
swap′ = [swap] ∅
```

4 Lambek-Grishin Calculus

The Lambek-Grishin calculus finds its origins in formal linguistics, where it is used to model natural language syntax. It is a symmetric version of the Lambek calculus, which means that in addition to left and right implication and conjunction, we have left and right difference (the operators dual to left and right implication) and disjunction (dual to conjunction). The basic inference rules for these (dual) connectives, together with a set of interaction principles between the connectives and their duals, allow for the treatment of patterns beyond the context-free, which cannot be satisfactorily handled in traditional Lambek calculus.

The formulation of the Lambek-Grishin calculus that we will model is the formulation developed in Moortgat and Moot (2013), which uses the mechanisms of polarity and focusing together with concepts from display logics to ensure, amongst others, that all proof terms are in normal form.

Below we will present a formalisation of **LG**, discussing the roles these mechanisms play in our model in turn.

Since this paper is not by far a complete discussion of the Lambek-Grishin calculus, we refer the interested reader to Moortgat and Moot (2013) or Bastenhof (2013).

4.1 Basic types and polarisation

The Lambek-Grishin calculus as developed in Moortgat and Moot (2013) is a polarised logic. Therefore, we will have to define a notion of polarity.

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

Using this definition, we can define our types as below.

```
data Type : Set where
  el    : (A : U) → (p : Polarity) → Type
  _⊗_  : Type → Type → Type
  _ \ _ : Type → Type → Type
  _ / _ : Type → Type → Type
  _⊕_  : Type → Type → Type
  _⊘_  : Type → Type → Type
  _⦸_  : Type → Type → Type
```

While the atomic types are assigned a polarity, the polarity of complex types is implicit in the connectives. We shall therefore define a pair of predicates that have inhabitants only if their argument is a positive or negative type.

```
data Pos : Type → Set where
  el : ∀ A → Pos (el A +)
  _⊗_ : ∀ A B → Pos (A ⊗ B)
  _⊘_ : ∀ B A → Pos (B ⊘ A)
  _⦸_ : ∀ A B → Pos (A ⦸ B)
```

```
data Neg : Type → Set where
  el : ∀ A → Neg (el A −)
  _⊕_ : ∀ A B → Neg (A ⊕ B)
  _ \ _ : ∀ A B → Neg (A \ B)
  _ / _ : ∀ B A → Neg (B / A)
```

We can trivially show that polarity is a decidable property, and that every type is either positive or negative.

```
Pol? : ∀ A → Pos A ⊎ Neg A
Pol? (el A +)  = inj₁ (el A)
Pol? (el A −)  = inj₂ (el A)
Pol? (A ⊗ B) = inj₁ (A ⊗ B)
Pol? (A ⊘ B) = inj₁ (A ⊘ B)
Pol? (A ⦸ B) = inj₁ (A ⦸ B)
Pol? (A ⊕ B) = inj₂ (A ⊕ B)
Pol? (A \ B) = inj₂ (A \ B)
Pol? (A / B) = inj₂ (A / B)
```

We also define Pos? and Neg?, which are decision procedures for the predicates Pos and Neg. Using these decision procedures, we can implicitly restrict the usage of inference rules to types of a certain polarity using a well-known Agda trick. For instance, the full type of the μ-rule (see §§ 4.3) is:

```
μ : ∀ {X A} {p : True (Neg? A)} → X ⊢ · A · → X ⊢ [A]
```

The idea behind this type is that, since we know that the decision procedure Neg? terminates, we can run it during type-checking to see if we can construct a witness of Neg A. If we can, True (Neg? A) reduces to to the unit type ⊤, and its value is trivially inferred; if we cannot, it reduces to the empty type ⊥—for which we know that we cannot construct an inhabitant—and a type-error is raised.[17]

4.2 Contexts and the display property

Since the Lambek-Grishin calculus is a display calculus, we will also have to model polarised structures (positive/input structures for the antecedent, negative/output structures for the succedent). In this case, the formulas that can appear as arguments to a connective are actually limited by their polarity, so we can encode the polarities at the type-level.

```
mutual
  data Struct⁺ : Set where
    ·_·   : Type → Struct⁺
    _⊗_   : Struct⁺ → Struct⁺ → Struct⁺
    _⊘_   : Struct⁺ → Struct⁻ → Struct⁺
    _⦸_   : Struct⁻ → Struct⁺ → Struct⁺
  data Struct⁻ : Set where
    ·_·   : Type → Struct⁻
    _⊕_   : Struct⁻ → Struct⁻ → Struct⁻
    _\_   : Struct⁺ → Struct⁻ → Struct⁻
    _/_   : Struct⁻ → Struct⁺ → Struct⁻
```

As a consequence of this, we do not have to bother with predicates for polarity in the case of structures, as a structure's polarity is immediately obvious from its type.

As **LG** is formulated as a display logic, we define a left and a right inference rule for each connective, where the one is a rule that simply structuralises the formula, and the other eliminates the connective when it appears as the outermost connective on both sides. Which is which depends on the polarity of the connective (i.e. on which side it naturally occurs). As an example, the left and right rules for _⊗_ are presented below.

```
⊗L : · A · ⊗ · B · ⊢ X → · A ⊗ B · ⊢ X
⊗R : X ⊢ [A] → Y ⊢ [B] → X ⊗ Y ⊢ [A ⊗ B]
```

4.3 Inference rules for focused sequents

Since **LG** is a focused calculus, and thus has several kinds of sequents, we will have to define its inference rules in several data types. Every inference rule is defined in the data type corresponding to the sequent-type of its conclusion.

Though it is a bit verbose, due to **LG**'s many inference rules, we would like to present the reader with the complete definition below.

[17] See http://agda.github.io/agda-stdlib/html/Relation.Nullary.Decidable.html#783 for the complete implementation.

Note we use the technique discussed in §§ 4.1 to restrict the applications of $\tilde{\mu}$ and $\mu*$ to the cases where A is positive and of μ and $\tilde{\mu}*$ those where A is negative. We will discuss the motives behind this in §§ 4.4.

```
mutual
  data _⊢[_] : Struct⁺ → Type → Set where
    var   : · A · ⊢ [A]
    μ     : X ⊢ · A · → X ⊢ [A]
    ⊗R    : X ⊢ [A] → Y ⊢ [B] → X ⊗ Y ⊢ [A ⊗ B]
    ⊘R    : X ⊢ [A] → [B] ⊢ Y → X ⊘ Y ⊢ [A ⊘ B]
    ⦸R    : [A] ⊢ X → Y ⊢ [B] → X ⦸ Y ⊢ [A ⦸ B]
  data [_]⊢_ : Type → Struct⁻ → Set where
    covar : [A] ⊢ · A ·
    μ̃     : · A · ⊢ X → [A] ⊢ X
    ⊕L    : [A] ⊢ Y → [B] ⊢ X → [A ⊕ B] ⊢ X ⊕ Y
    \L    : X ⊢ [A] → [B] ⊢ Y → [A \ B] ⊢ X \ Y
    /L    : [A] ⊢ Y → X ⊢ [B] → [A / B] ⊢ Y / X
  data _⊢_ : Struct⁺ → Struct⁻ → Set where
    μ*    : X ⊢ [A] → X ⊢ · A ·
    μ̃*    : [A] ⊢ X → · A · ⊢ X
    ⊗L    : · A · ⊗ · B · ⊢ X → · A ⊗ B · ⊢ X
    ⊘L    : · A · ⊘ · B · ⊢ X → · A ⊘ B · ⊢ X
    ⦸L    : · A · ⦸ · B · ⊢ X → · A ⦸ B · ⊢ X
    ⊕R    : X ⊢ · A · ⊕ · B · → X ⊢ · A ⊕ B ·
    \R    : X ⊢ · A · \ · B · → X ⊢ · A \ B ·
    /R    : X ⊢ · A · / · B · → X ⊢ · A / B ·
    res₁  : Y ⊢ X \ Z → X ⊗ Y ⊢ Z
    res₂  : X ⊗ Y ⊢ Z → Y ⊢ X \ Z
    res₃  : X ⊢ Z / Y → X ⊗ Y ⊢ Z
    res₄  : X ⊗ Y ⊢ Z → X ⊢ Z / Y
    dres₁ : Z ⊘ X ⊢ Y → Z ⊢ Y ⊕ X
    dres₂ : Z ⊢ Y ⊕ X → Z ⊘ X ⊢ Y
    dres₃ : Y ⦸ Z ⊢ X → Z ⊢ Y ⊕ X
    dres₄ : Z ⊢ Y ⊕ X → Y ⦸ Z ⊢ X
    dist₁ : X ⊗ Y ⊢ Z ⊕ W → X ⊘ W ⊢ Z / Y
    dist₂ : X ⊗ Y ⊢ Z ⊕ W → Y ⊘ W ⊢ X \ Z
    dist₃ : X ⊗ Y ⊢ Z ⊕ W → Z ⦸ X ⊢ W / Y
    dist₄ : X ⊗ Y ⊢ Z ⊕ W → Z ⦸ Y ⊢ X \ W
```

As we have dropped exchange, we will have to let go of our running example. Instead we will present the proof of the law of raising, and its dual law of lowering.

```
raise : · A · ⊢ · (B / A) \ B ·
raise = \R (res₂ (res₃ (μ̃* (/L covar var))))
lower : · B ⊘ (A ⦸ B) · ⊢ · A ·
lower = ⊘L (dres₂ (dres₃ (μ* (⦸R covar var))))
```

4.4 Reification into LP

Finally, we will present the reification of **LG** terms into **LP**, as we did for **LP** to **IL** in §§ 3.2. Since **LG** is a classical logic, in the sense that every connective has a dual, we cannot give it a direct interpretation in the intuitionistic **LP**.

The term language of **LG**, however, is a refinement of the $\bar{\lambda}\mu\tilde{\mu}$-calculus as developed in Curien and Herbelin (2000), which has a known computational interpretation through translation to Parigot's $\lambda\mu$-calculus. Therefore, we can give an interpretation through a CPS-translation.

Below we formalise the CPS-translation of **LG** as presented in Moortgat and Moot (2013). The idea of the CPS-translation is to interpret all connectives as a conjunctions, and use the polarities of the connectives and their argument positions to guide the introduction of negations—when the natural polarity of an argument position clashes with the polarity of its inhabitant, we lift the inhabitant's type to a continuation.

```
mutual
  ⟦_⟧⁺ : Type → Type^LP
  ⟦ el A + ⟧⁺ = el A
  ⟦ el A − ⟧⁺ = ¬ (¬ el A)
  ⟦ A ⊗ B ⟧⁺ =    ⟦ A ⟧⁺ ⊗ ⟦ B ⟧⁺
  ⟦ A ⊘ B ⟧⁺ =    ⟦ A ⟧⁺ ⊗ ⟦ B ⟧⁻
  ⟦ A ⦸ B ⟧⁺ =    ⟦ A ⟧⁻ ⊗ ⟦ B ⟧⁺
  ⟦ A ⊕ B ⟧⁺ = ¬ (⟦ A ⟧⁻ ⊗ ⟦ B ⟧⁻)
  ⟦ A \ B ⟧⁺ = ¬ (⟦ A ⟧⁺ ⊗ ⟦ B ⟧⁻)
  ⟦ A / B ⟧⁺ = ¬ (⟦ A ⟧⁻ ⊗ ⟦ B ⟧⁺)
  ⟦_⟧⁻ : Type → Type^LP
  ⟦ el A + ⟧⁻ = ¬ el A
  ⟦ el A − ⟧⁻ = ¬ el A
  ⟦ A ⊗ B ⟧⁻ = ¬ (⟦ A ⟧⁺ ⊗ ⟦ B ⟧⁺)
  ⟦ A ⊘ B ⟧⁻ = ¬ (⟦ A ⟧⁺ ⊗ ⟦ B ⟧⁻)
  ⟦ A ⦸ B ⟧⁻ = ¬ (⟦ A ⟧⁻ ⊗ ⟦ B ⟧⁺)
  ⟦ A ⊕ B ⟧⁻ =    ⟦ A ⟧⁻ ⊗ ⟦ B ⟧⁻
  ⟦ A \ B ⟧⁻ =    ⟦ A ⟧⁺ ⊗ ⟦ B ⟧⁻
  ⟦ A / B ⟧⁻ =    ⟦ A ⟧⁻ ⊗ ⟦ B ⟧⁺
```

In addition to this, we define a CPS-translation of positive and negative structures. This translation is much the same, but since the structures' types enforce that no clashes in polarity occur, no negations are introduced.

Below we will present an implementation of the reification function up to exchange (that is, we do not show applications of the exchange principle).

As we stated above, we interpret all connectives as pairs. Therefore, all left and right rules are interpreted as ⊗-introduction or ⊗-elimination.

The μ- and $\tilde{\mu}$-rules are translated as lambda abstractions, and the $\mu*$ and $\tilde{\mu}$-rules are translated as applications. However, if you look closely at the types, you will notice that the sequents μ and $\tilde{\mu}$ are translated to do not necessarily derive function types—and neither do the sequents for the first arguments of $\mu*$ or $\tilde{\mu}$. This is where the polarity restrictions come in: we restrict the application of these rules to a certain polarity, so we can later use this fact to prove that a clash in polarities *must* occur during the CPS-translation, and therefore that a function-type *must* be generated, using the following lemmas.

```
Neg-≡ : Neg A → ⟦ A ⟧⁺ ≡ ⟦ A ⟧⁻ ⊸ ⊥
Pos-≡ : Pos A → ⟦ A ⟧⁻ ≡ ⟦ A ⟧⁺ ⊸ ⊥
```

Lastly, variables and co-variables are both simply translated as variables.[18]

```
mutual
  reifyʳ : ∀ {X A} → X ⊢ [A] → ⟦ X ⟧ ⊢LP ⟦ A ⟧⁺
  reifyʳ var         = var
  reifyʳ (μ t)       rewrite Neg-≡ _ = abs (reify t)
  reifyʳ (⊗R s t)    = pair (reifyʳ s) (reifyʳ t)
  reifyʳ (⊘R s t)    = pair (reifyʳ s) (reifyˡ t)
  reifyʳ (⦸R s t)    = pair (reifyˡ s) (reifyʳ t)

  reifyˡ : ∀ {A Y} → [A] ⊢ Y → ⟦ Y ⟧ ⊢LP ⟦ A ⟧⁻
  reifyˡ covar       = var
  reifyˡ (μ̃ t)       rewrite Pos-≡ _ = abs (reify t)
  reifyˡ (⊕L s t)    = pair (reifyˡ s) (reifyˡ t)
  reifyˡ ( \ L s t)  = pair (reifyʳ s) (reifyˡ t)
  reifyˡ ( / L s t)  = pair (reifyˡ s) (reifyʳ t)

  reify : ∀ {X Y} → X ⊢ Y → ⟦ X ⟧ ++ ⟦ Y ⟧ ⊢LP ⊥
  reify (μ* t)       rewrite Pos-≡ _ = app var (reifyʳ t)
  reify (μ̃* t)       rewrite Neg-≡ _ = app var (reifyˡ t)
  reify (⊗L t)       = case var (reify t)
  reify (⊘L t)       = case var (reify t)
  reify (⦸L t)       = case var (reify t)
  reify (⊕R t)       = case var (reify t)
  reify ( \ R t)     = case var (reify t)
  reify ( / R t)     = case var (reify t)
```

Using the above definition, we can now define the function for CPS-interpretation of **LG** by composition.

```
[_] : X ⊢ [A] → (Ctxt ⟦ X ⟧ → ⟦ A ⟧)
[_] = [_]^LP ∘ reifyʳ
```

4.5 Examples from natural language

In this final section we will demonstrate a possible usage of our model of **LG**. We will derive the denotation of the following example sentence.

[18] In Moortgat and Moot (2013), applications of the var and covar rules is also limited to certain polarities. However, we do not need this fact to implement the CPS-translation, and therefore we have chosen not to add this restriction. A benefit of this is that defining derived inference rules becomes much more manageable, as we no longer have to ensure that the used variables have a certain polarity.

"Everyone finds some unicorn."

First, we define a couple of meaning postulates. For brevity's sake, we will also postulate the existence of an Entity type with the corresponding universal and existential quantifiers, though we could trivially define this ourselves.

```
postulate
  Entity   : Set
  _⊃_      : Bool → Bool → Bool
  FORALL   : (Entity → Bool) → Bool
  EXISTS   : (Entity → Bool) → Bool
  PERSON   : Entity → Bool
  FIND     : Entity → Entity → Bool
  UNICORN  : Entity → Bool
```

Secondly, we must define our type universe. In this case we will define it to be the usual set of atomic syntactic types.

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

⟦_⟧ᵘ  : U → Set
⟦ S  ⟧ᵘ = Bool
⟦ N  ⟧ᵘ = Entity → Bool
⟦ NP ⟧ᵘ = Entity
```

Next, we define our lexicon. The entries of our lexicon are lambda terms typed by the translations of their syntactic types.

```
everyone : ⟦ (el NP + / el N +) ⊗ el N + ⟧
everyone = ((λ {(A, B) → FORALL (λ x → B x ⊃ A x)}), PERSON)
finds    : ⟦ (el NP + \ el S −) / el NP + ⟧
finds    = λ {((x, k), y) → k (FIND y x)}
some     : ⟦ el NP + / el N + ⟧
some     = λ {(A, B) → EXISTS (λ x → A x ∧ B x)}
unicorn  : ⟦ el N + ⟧
unicorn  = UNICORN
```

Last, since we have not yet proven decidability, we have to give a proof that our sentence structure is syntactically correct.

```
sent :
  · (el NP + / el N +) ⊗ el N +          -- everyone
  · ⊗ (· (el NP + \ el S −) / el NP +    -- finds
  · ⊗ (· el NP + / el N +                -- some
  · ⊗ · el N + ·                         -- unicorn
  )) ⊢ [el S −]
sent =
  μ (res₃ (⊗L (res₃ (μ̃* ( / L (
    μ̃ (res₄ (res₁ (res₁ (res₃ (μ̃* ( / L (
      μ̃ (res₂ (res₃ (μ̃* ( / L ( \ L var covar) var))))) var))))))) var)))))
```

With all these components, we can finally compute the meaning of our sentence, leaving our meaning postulates unevaluated as usual.

```
[sent] (everyone, finds, some, unicorn, ∅) ⇝β
  λ k → FORALL (λ x₁ → PERSON x₁ ⊃ EXISTS (λ x₂ → k (FIND x₂ x₁) ∧ UNICORN x₂))
```

5 Future work

Reification of properties. When we reify a term in a substructural logic into Agda, we lose the information regarding its behaviour. For instance, if we have an proof in the presented model of **LP**, we would like to be able to obtain a proof of linearity for the reified term.

Decidability of focus shifting. If we could implement a decision procedure for the focus shifting principles (not discussed in this paper; a sequence of unfocused rules, started by a μ-application and terminated by a μ-abstraction), we could add them as derived rules to our model of **LG**. This would make writing proofs much easier, and would be a good step in the direction of proving decidability of **LG** in general.

Decidability of LG. We could implement a decision procedure for **LG** in general. Using this procedure we would no longer have to manually prove syntactic correctness. In addition to this, if we implemented decidability of **LG** plus associativity, we could use the resulting procedure as an implementation of parsing-as-deduction.

Mirror symmetries. Another property of **LG** is that types and proofs obey certain mirror symmetries (due to the presence of dual operators and directional implications). Implementing these symmetries as functions on types and proofs would allow us to easily construct the duals of types and their proofs, and would aid in the understanding of these dualities.

Extract Haskell library. Since Agda supports the extraction of programs into several languages (most notably Haskell and JavaScript) we could investigate the extraction of an optimised Haskell library for **LG** (and its use in natural language processing) from our implementation.

6 Conclusion

We have presented the reader with several models of intuitionistic logic, and examined several models for substructural logics (linear logic and the Lambek-Grishin calculus). We have shown how proofs in these models can be given an interpretation in Agda through reification and translation. And, last, we have demonstrated the usage of our models in an example taken from formal linguistics.

References

Andreoli, J.-M. (1992). Logic programming with focusing proofs in linear logic. *Journal of Logic and Computation*, 2:297–347.

Atkey, R. (2006). Parameterized Notions of Computation. In *MSFP 2006*.

Bastenhof, A. (2012). Polarized montagovian semantics for the lambek-grishin calculus. In *Proceedings of the 15th and 16th International Conference on Formal Grammar*, FG'10/FG'11, pages 1–16, Berlin, Heidelberg. Springer-Verlag.

Bastenhof, A. (2013). Categorial symmetry.

Belnap, N. D. (1982). Display logic. *Journal of Philosophical Logic*, 11(4):375–417.

Curien, P.-L. and Herbelin, H. (2000). The duality of computation. *SIGPLAN Not.*, 35(9):233–243.

de Bruijn, N. G. (1972). Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *INDAG. MATH*, 34:381–392.

de Groote, P. (1994). A cps-translation of the $\lambda\mu$-calculus.

de Groote, P. (2001). Towards abstract categorial grammars. In *Association for Computational Linguistics, 39th Annual Meeting and 10th Conference of the European Chapter, Proceedings of the Conference*, pages 148–155.

Érdi, G. (2013). Simply typed lambda calculus in agda, without shortcuts. http://gergo.erdi.hu/blog/2013-05-01-simply_typed_lambda_calculus_in_agda,_without_shortcuts/. Accessed: 2014-01-28.

Girard, J.-Y. (1991). A new constructive logic: classic logic. *Mathematical Structures in Computer Science*, 1:255–296.

Kiselyov, O., Lämmel, R., and Schupke, K. (2004). Strongly typed heterogeneous collections. In *Proceedings of the 2004 ACM SIGPLAN Workshop on Haskell*, Haskell '04, pages 96–107, New York, NY, USA. ACM.

Lindblad, F. and Benke, M. (2006). A tool for automated theorem proving in agda. In *Proceedings of the 2004 International Conference on Types for Proofs and Programs*, TYPES'04, pages 154–169, Berlin, Heidelberg. Springer-Verlag.

Martin-Löf, P. (1984). Intuitionistic type theory.

Mazzoli, F. (2013). Agda by example: λ-calculus. http://mazzo.li/posts/Lambda.html. Accessed: 2014-01-28.

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

Moortgat, M. and Moot, R. (2013). Proofs nets and the categorial flow of information.

Mu, S.-C. (2008). Typed λ-calculus interpreter in agda. http://www.iis.sinica.edu.tw/~scm/2008/typed-lambda-calculus-interprete/. Accessed: 2014-01-28.

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.

Parigot, M. (1992). λμ-calculus: An algorithmic interpretation of classical natural deduction. In Voronkov, A., editor, *Logic Programming and Automated Reasoning*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer Berlin Heidelberg.

Restall, G. (2011). Substructural logics. In Zalta, E. N., editor, *The Stanford Encyclopedia of Philosophy*. Summer 2011 edition.

Wadler, P. L. (1993). A Taste of Linear Logic. In *Proceedings of the 18th International Symposium on Mathematical Foundations of Computer Science, Gdánsk*, New York, NY. Springer-Verlag.