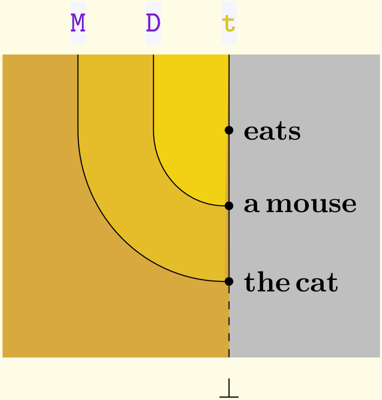
# Formalizing Typing Rules for Natural Languages using Effects

Matthieu Boyer

18th April 2025 at CLaY

# Plan

# What the hell am I doing?



# Defining definites and indefinites

A semantic theory for natural language is generally defined by three components:

- A syntax, for the generation of grammatical expressions, which we will not really discuss

## Defining definites and indefinites

A semantic theory for natural language is generally defined by three components:

- A syntax, for the generation of grammatical expressions, which we will not really discuss
- A lexicon, for defining the meanings of individual expressions, which we will discuss by necessity

# Defining definites and indefinites

A semantic theory for natural language is generally defined by three components:

- A syntax, for the generation of grammatical expressions, which we will not really discuss
- A lexicon, for defining the meanings of individual expressions, which we will discuss by necessity
- A theory of composition, describing how the meanings of complex expressions are built from their parts.

## Defining definites and indefinites

A semantic theory for natural language is generally defined by three components:

- A syntax, for the generation of grammatical expressions, which we will not really discuss
- A lexicon, for defining the meanings of individual expressions, which we will discuss by necessity
- A theory of composition, describing how the meanings of complex expressions are built from their parts.

We will call the pair (syntax, lexicon) the language.

## Defining definites and indefinites

However this generates new difficulties:

(1) Every planet shines.

(2) A planet shines.

(3) No planet shines.

This means in particular that **Every**, **A** and **No** should all have the same type `e` since **shines** has type `e` $\rightarrow$ `t`.

# Defining definites and indefinites

However this generates new difficulties:

(4) Every planet shines.

(5) A planet shines.

(6) No planet shines.

This means in particular that **Every**, **A** and **No** should all have the same type $\mathtt{e}$ since **shines** has type $\mathtt{e} \to \mathtt{t}$. We will in this presentation provide a computable extension of usual theories of composition, based on a type and effect system to avoid this issue, based on [**?**].

## Plan

# Plan

# Slide 42

**Definition 2.1 — Category.** A (small) *category* is described by the following data:

0 A class of objects (nodes of a graph).

# Slide 42

**Definition 2.2 — Category.** A (small) *category* is described by the following data:

0 A class of objects (nodes of a graph).

1 For a pair $A, B$ of objects, a set $\mathrm{Hom}(A, B)$ of functions from $A$ to $B$ called *morphisms*, *maps* or *arrows*. We denote it by $f : A \to B$ or $A \xrightarrow{f} B$.

# Slide 42

**Definition 2.3 — Category.** A (small) *category* is described by the following data:

0 A class of objects (nodes of a graph).

1 For a pair $A, B$ of objects, a set $\mathrm{Hom}(A, B)$ of functions from $A$ to $B$ called *morphisms*, *maps* or *arrows*. We denote it by $f : A \to B$ or $A \xrightarrow{f} B$.

2 For all triplets $A, B, C$, a composition law $\circ_{A,B,C}$:

$$\begin{array}{rcl} \mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) & \to & \mathrm{Hom}(A, C) \\ (g, f) & \mapsto & g \circ f \end{array}$$

# Slide 42

**Definition 2.4 — Category.** A (small) *category* is described by the following data:

0 A class of objects (nodes of a graph).

1 For a pair $A, B$ of objects, a set $\mathrm{Hom}(A, B)$ of functions from $A$ to $B$ called *morphisms*, *maps* or *arrows*. We denote it by $f : A \to B$ or $A \xrightarrow{f} B$.

2 For all triplets $A, B, C$, a composition law $\circ_{A,B,C}$:

$$\begin{array}{rcl} \mathrm{Hom}(B, C) \times \mathrm{Hom}(A, B) & \to & \mathrm{Hom}(A, C) \\ (g, f) & \mapsto & g \circ f \end{array}$$

2 For all objects $A$, an identity map $\mathrm{id}_A \in \mathrm{Hom}(A, A)$.

# Slide 42

**Definition 2.5 — Category.** A (small) *category* is described by the following data:

3 Associativity: $f \circ (g \circ h) = (f \circ g) \circ h = f \circ g \circ h$

# Slide 42

**Definition 2.6 — Category.** A (small) *category* is described by the following data:

3 Associativity: $f \circ (g \circ h) = (f \circ g) \circ h = f \circ g \circ h$

3 Unitarity: $f \circ \mathrm{id}_A = f = \mathrm{id}_B \circ f$.

# Slide 42

A few examples of categories are:

**Set** whose objects are Sets and arrows are functions between sets.

# Slide 42

A few examples of categories are:

**Set** whose objects are Sets and arrows are functions between sets.

**Top** whose objects are Topological Spaces and arrows are continuous maps between those.

# Slide 42

A few examples of categories are:

- **Set** whose objects are Sets and arrows are functions between sets.
- **Top** whose objects are Topological Spaces and arrows are continuous maps between those.
- **Grp** whose objects are Groups and arrows are Group Homomorphisms.

# Slide 42

A few examples of categories are:

- **Set** whose objects are Sets and arrows are functions between sets.
- **Top** whose objects are Topological Spaces and arrows are continuous maps between those.
- **Grp** whose objects are Groups and arrows are Group Homomorphisms.
- **Vec** whose objects are Vector Spaces on a field $k$ and arrows are Linear Maps.

# Commuter Rail, basically

The right language for categories is the commutative diagram one.

The associativity rewrites as:

$$
\begin{tikzcd}
 & B \arrow[r, "g"] & C \arrow[dr, "h"] & \\
A \arrow[ur, "f"] \arrow[urr, "g\circ f"'] \arrow[rrr, "h\circ(g\circ f)"'] & & & D
\end{tikzcd}
$$

# Commuter Rail, basically

In $Set$ we have the following commutative diagram:

$$
\begin{tikzcd}
\mathbb{R} \arrow[r, "\times 2"] \arrow[d, ".^2"'] & \mathbb{R} \arrow[d, ".^2"] \\
\mathbb{R}^+ \arrow[r, "\times 4"] & \mathbb{R}^+
\end{tikzcd}
$$

# Commuter Rail, basically

In $Set$ we have the following commutative diagram:

$$\begin{tikzcd} \mathbb{R} \arrow[r, "\times 2"] \arrow[d, ".^2"'] & \mathbb{R} \arrow[d, ".^2"] \\ \mathbb{R}^+ \arrow[r, "\times 4"] & \mathbb{R}^+ \end{tikzcd}$$

It simply states that:

$$\forall x \in \mathbb{R}, \left(2x^2\right) = 4x^2$$

# Of wolf, and man.

**Definition 2.7** Let $\mathcal{A}, \mathcal{B}$ be two categories. A functor $\mathcal{F} : \mathcal{A} \to \mathcal{B}$ is:

0 An object $F(A) \in \mathcal{B}$ for each object $A$ of $\mathcal{A}$.

1 For each pair $A_1, A_2 \in \mathcal{A}$, a function:

$$\begin{array}{rrcl} F_{A_1,A_2} : & \mathrm{Hom}_{\mathcal{A}}(A_1, A_2) & \to & \mathrm{Hom}_{\mathcal{B}}(FA_1, FA_2) \\ & f & \mapsto & F(f) \end{array}$$

# Of wolf, and man.

We ask the following equations to be satisfied:

$F(g \circ f) = F(g) \circ F(f)$, that is

$$\begin{tikzcd} A \arrow[r, "g"] \arrow[d, "F"] & B \arrow[r, "f"] \arrow[d, "F"] & C \arrow[d, "F"] \\ FA \arrow[r, "Fg"'] & FB \arrow[r, "Ff"'] & FC \end{tikzcd}$$

and $F(\mathrm{id}_A) = \mathrm{id}_{F(A)}$

# O.K. Corral

**Definition 2.8** A natural transformation is a functor in the category of small categories and functors. If $F, G : \mathcal{A} \Rightarrow \mathcal{B}$ are functors, a natural transformation $\theta$ from $F$ to $G$ is, for each object of $\mathcal{A}$ a function $\theta_A$ such that the following diagram commutes for all $f : A \to B$.

$$\begin{tikzcd} FA \arrow[r, "Ff"] \arrow[d, "\theta_A"'] & FB \arrow[d, "\theta_B"] \\ GA \arrow[r, "Gf"] & GB \end{tikzcd}$$

# O.K. Corral

**Definition 2.9** An adjunction $L \dashv R$ between two functors $L : \mathcal{A} \to \mathcal{B}$ and $R : \mathcal{B} \to \mathcal{A}$ is a pair of natural transformations $\eta : \mathrm{Id}_{\mathcal{A}} \Rightarrow R \circ L$ and $\varepsilon : L \circ R \Rightarrow \mathrm{Id}_{\mathcal{B}}$ verifying the zigzag equations:

$$
\begin{tikzcd}
\mathcal{A} \arrow[r, "\mathrm{id}_{\mathcal{A}}"] \arrow[dr, "L"'] \arrow[r, phantom, shift right=6ex, "\boxed{\eta}"] & \mathcal{A} \arrow[dr, "L"] \arrow[dr, phantom, shift left=2ex, "\varepsilon"] & \\
& \mathcal{B} \arrow[u, "R"] \arrow[r, "\mathrm{id}_B"'] & \mathcal{B}
\end{tikzcd}
\quad = \quad
\begin{tikzcd}
\mathcal{A} \arrow[r, bend left, "L"] \arrow[r, bend right, "L"'] \arrow[r, phantom, "\Downarrow \mathrm{id}_L"] & \mathcal{B}
\end{tikzcd}
$$

# Plan

# Yu-Gi-Oh

**Definition 2.10** A product of two objects $A$ and $B$ in a category $\varphi$ is a triplet

$$(A \times B, \pi_1 : A \times B \to A, \pi_2 : A \times B \to B)$$

$$\begin{tikzcd}
A & & B \\
& A \times B \arrow[ul, "\pi_1"] \arrow[ur, "\pi_2"'] &
\end{tikzcd}$$

such that for all pair of arrows $X \xrightarrow{f} A$ et $X \xrightarrow{g} B$, there is a unique $h : X \to A \times B$ such that $f = \pi_1 \circ h, g = \pi_2 \circ h$.

# Magic

**Definition 2.11** A terminal object $\mathbb{1}$ in a category $\Gamma \vdash$ is an object such that for all $A$ in $\Gamma \vdash$ there is one and only one arrow $A \to \mathbb{1}$.

An initial object is the same thing with the arrows reversed.

**Definition 2.12** A category is cartesian if all products exist and it has a terminal object $\mathbb{1}$.

Cartesian products and terminal objects are unique, up to isomorphism.

# Go Fish

**Definition 2.13** A cartesian closed category is a cartesian category where we define for each object $A$ a functor $A \Rightarrow \Gamma \vdash \to \Gamma \vdash$ right adjunct to $A \times \Gamma \vdash \to \Gamma \vdash$. This means we have bijections $\Phi_{X,Y} : \mathrm{Hom}\,(A \times X, Y) \to \mathrm{Hom}\,(X, A \Rightarrow Y)$ called currification bijections..

# Go Fish

**Definition 2.14** A cartesian closed category is a cartesian category where we define for each object $A$ a functor $A \Rightarrow \Gamma \vdash \to \Gamma \vdash$ right adjunct to $A \times \Gamma \vdash \to \Gamma \vdash$. This means we have bijections $\Phi_{X,Y} : \mathrm{Hom}\,(A \times X, Y) \to \mathrm{Hom}\,(X, A \Rightarrow Y)$ called currification bijections..

$Set$ is a cartesian closed category, where currification is defined by the partial application of high arity functions to a subset of their arguments.

# Use the Types, Luke

**Definition 2.15** The $\lambda$-calculus is defined by the following basic grammar:

$$
\begin{array}{rcll}
\mathsf{E} & ::= & x \in Var & \textit{(Variables)} \\
 & \mid & App(E, E) & \textit{(Application)} \\
 & \mid & \lambda x.E & \textit{(Evaluation)}
\end{array}
$$

# Use the Types, Luke

**Definition 2.16** We give ourselves a set of type variables $TyVar$ and define types by:

$$
\begin{array}{rcl}
A, B & ::= & \alpha \in TyVar \\
& \mid & A \times B \\
& \mid & \mathbb{1} \\
& \mid & A \Rightarrow B
\end{array}
$$

A context $M = x_1 : A_1, \ldots, x_n : A_n$ is a list of pairs $x_i : A_i$ with a variable $x_i$ and a type $A_i$, all the variables being different.

# Fudge Supreme

**Definition 2.17** A typing $\Gamma \vdash M : A$ is a triplet composed of a context $\Gamma$, a $\lambda$-term $M$ and a type $A$, such that all free variables of $M$ are in $\Gamma$. A proof tree for a typing judgement is constructed inductively from a set of rules of the form:

$$\frac{}{x : A \vdash x : A}\ \text{Var}$$

$$\frac{\Gamma \vdash M : A \Rightarrow B \qquad \Delta \vdash N : A}{\Gamma, \Delta \vdash App(M, N) : B}\ \text{App}$$

# Abstract Concrete

We will place ourselves in the functional programming paradigm, where everything is a function. Let $(\mathcal{C}, \times, \perp, \Rightarrow)$ be a cartesian closed category. The type variables are the object of our category.

## Abstract Concrete

We will place ourselves in the functional programming paradigm, where everything is a function. Let $(\mathcal{C}, \times, \perp, \Rightarrow)$ be a cartesian closed category. The type variables are the object of our category.The type of an arrow $A \xrightarrow{f} B$ is $A \Rightarrow B \in \mathcal{C}$. For constants of type $A$ in our language, we can also say their type is $\perp \to A$ to keep the function interpretation.

# Plan

# Plan

## Underfull hbox

A side-effect is something that results from a computation without it being its actual return value. It could be:

# Underfull hbox

A side-effect is something that results from a computation without it being its actual return value. It could be:

- The printing of a value to the console;

## Underfull hbox

A side-effect is something that results from a computation without it being its actual return value. It could be:

- The printing of a value to the console;
- The modification of a variable in memory;

## Underfull hbox

A side-effect is something that results from a computation without it being its actual return value. It could be:

- The printing of a value to the console;
- The modification of a variable in memory;
- Non-determinism in the result.

# Underfull hbox

A side-effect is something that results from a computation without it being its actual return value. It could be:

- The printing of a value to the console;
- The modification of a variable in memory;
- Non-determinism in the result.

A way to model effects when in a typing category is by using functors [?].

# Liberty

Let $\mathcal{L}$ be our language:

- $\mathcal{O}\left(\mathcal{L}\right)$ is the set of words in the language whose semantic representation is a function.
- $\mathcal{F}\left(\mathcal{L}\right)$ the set of words whose semantic representation is a functor.

Let $\mathcal{C}$ be a cartesian closed category adapted for our language.

# Liberty

Let $\mathcal{L}$ be our language:

- $\mathcal{O}(\mathcal{L})$ is the set of words in the language whose semantic representation is a function.
- $\mathcal{F}(\mathcal{L})$ the set of words whose semantic representation is a functor.

Let $\mathcal{C}$ be a cartesian closed category adapted for our language.
Let $\bar{\mathcal{C}}$ the free categorical closure of $\mathcal{F}(\mathcal{L})(\mathcal{O}(\mathcal{L}))$.
Then our types are defined as: $\star = \mathrm{Obj}\left(\bar{\mathcal{C}}\right)/\mathcal{F}(\mathcal{L})$.
Let $\star_0 = \mathrm{Obj}(\mathcal{C})$.
We get a subtyping relationship based on the procedure used to construct $\bar{\mathcal{C}}$.

# Exactly

In $\bar{\mathcal{C}}$ a functor is a polymorphic function $\tau \in S \subseteq \star \to F\tau$.

# Exactly

In $\bar{\mathcal{C}}$ a functor is a polymorphic function $\tau \in S \subseteq \star \to F\tau$.

$$\frac{\Gamma \vdash x : \tau \in \star_0}{\Gamma \vdash Fx : F\tau \notin \star_0}\ \textsf{Func}_0 \qquad \frac{\Gamma \vdash x : \tau}{\Gamma \vdash Fx : F\tau \preceq \tau}\ \textsf{Func}$$

# Theorems for Free!

Remember that if we have a natural transformation $F \overset{\theta}{\Longrightarrow} G$ then for all arrows $f : \tau_1 \to \tau_2$ we have:

$$
\begin{tikzcd}
F\tau_1 \arrow[r, "Ff"] \arrow[d, "\theta_{\tau_1}"'] & F\tau_2 \arrow[d, "\theta_{\tau_2}"] \\
G\tau_1 \arrow[r, "Gf"] & G\tau_2
\end{tikzcd}
$$

# Theorems for Free!

Remember that if we have a natural transformation $F \overset{\theta}{\Longrightarrow} G$ then for all arrows $f : \tau_1 \to \tau_2$ we have:

$$\begin{tikzcd} F\tau_1 \arrow[r, "Ff"] \arrow[d, "\theta_{\tau_1}"'] & F\tau_2 \arrow[d, "\theta_{\tau_2}"] \\ G\tau_1 \arrow[r, "Gf"] & G\tau_2 \end{tikzcd}$$

Since it's true for all maps: from a proof of $\Gamma \vdash x : F\tau$ we should be able to prove $\Gamma \vdash x : G\tau$.

# Theorems for Free!

Remember that if we have a natural transformation $F \overset{\theta}{\Longrightarrow} G$ then for all arrows $f : \tau_1 \to \tau_2$ we have:

$$\begin{tikzcd}
F\tau_1 \arrow[r, "Ff"] \arrow[d, "\theta_{\tau_1}"'] & F\tau_2 \arrow[d, "\theta_{\tau_2}"] \\
G\tau_1 \arrow[r, "Gf"] & G\tau_2
\end{tikzcd}$$

Since it's true for all maps: from a proof of $\Gamma \vdash x : F\tau$ we should be able to prove $\Gamma \vdash x : G\tau$.

In the Haskell programming language, any polymorphic function is a natural transformation from the first type constructor to the second type constructor [**?**].

# Applicatives, Monads

Applicative Functors and Monads are special type of functors which allow for more composition rules. Let $F$ be a functor:

# Applicatives, Monads

Applicative Functors and Monads are special type of functors which allow for more composition rules. Let $F$ be a functor:

- $F$ if an applicative if there is a natural transformation $\eta : \mathrm{Id} \Rightarrow F$, this allows for the application of $F\,(a \to b)$ to an object of type $a$.

# Applicatives, Monads

Applicative Functors and Monads are special type of functors which allow for more composition rules. Let $F$ be a functor:

- $F$ if an applicative if there is a natural transformation $\eta : \mathrm{Id} \Rightarrow F$, this allows for the application of $F\,(a \to b)$ to an object of type $a$.
- $F$ is a monad if it is an applicative and there is a natural transformation $\mu : FF \Rightarrow F$ allowing for $a \to Fb$ applied to an object of type $Fa$.

# Applicatives, Monads

This basically create two new typing judgements:

$$\frac{\Gamma \vdash x : A\tau_1 \qquad \Gamma \vdash \varphi : A\left(\tau_1 \to \tau_2\right)}{\Gamma \vdash \varphi x : A\tau_2} \quad \texttt{<*>}$$

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : A\tau} \quad \texttt{pure/return} \qquad\qquad \frac{\Gamma \vdash x : MM\tau}{\Gamma \vdash x : M\tau} \quad \texttt{>>=}$$

# People are persons

An adjunction $L \dashv R$ also grants a typing rule from its existence since it gives a natural transformation $L \circ R \Rightarrow \mathrm{Id}$.

# People are persons

An adjunction $L \dashv R$ also grants a typing rule from its existence since it gives a natural transformation $L \circ R \Rightarrow \mathrm{Id}$. The power of our formalism is that it is easy to design higher-order constructs as effects who quasi-commute with other effects: Given a denotation for the plural we could easily create a functor that adds it to any noun/verb and adding natural transformations $F \circ \Pi \Rightarrow \Pi \circ F$.

# Plan

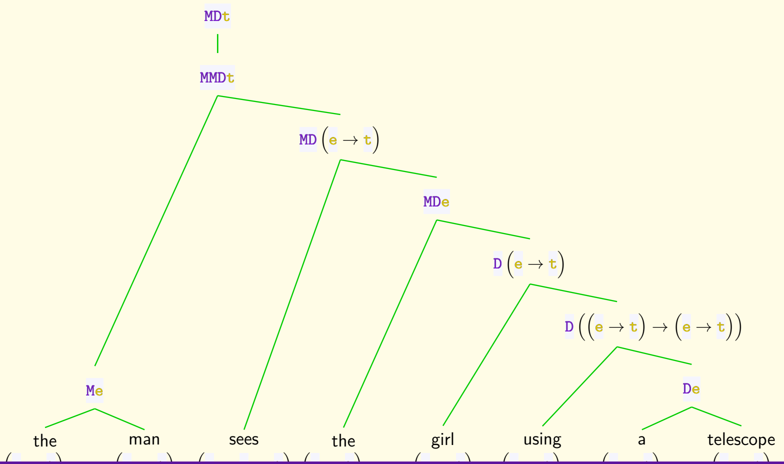
# Stick

An effect handler is a way to delete the additional things that yielded from the computation.
Formally it's a natural transformation $F \Rightarrow \mathrm{Id}$.

However, since everyone has a different way to handle non-determinism for example, we make handlers speaker-dependent instead of language-dependent.
In the following sections, we will consider a set set of handlers for our effects.

# Quantum

There is ambiguity in typing reductions for effects:



# Scarves

A string diagram is the Poincaré Dual of a categorical diagram in $Cat$:

# Scarves

A string diagram is the Poincaré Dual of a categorical diagram in $Cat$:

- Regions are different categories.

# Scarves

A string diagram is the Poincaré Dual of a categorical diagram in $Cat$:

- Regions are different categories.
- Borders are functors.

# Scarves

A string diagram is the Poincaré Dual of a categorical diagram in $Cat$:

- Regions are different categories.
- Borders are functors.
- Dots are natural transformations.

# Scarves

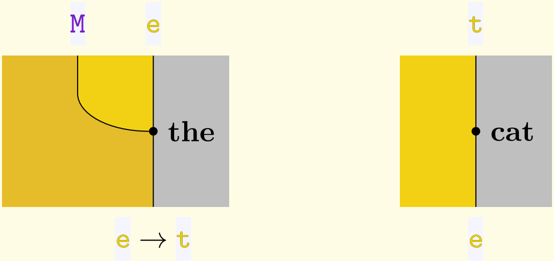
A string diagram is the Poincaré Dual of a categorical diagram in $Cat$:

- Regions are different categories.
- Borders are functors.
- Dots are natural transformations.

We read a diagram right to left, bottom to top.

# Scarves

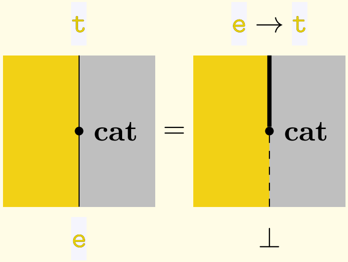
Considering the category $\mathbb{1}$ with a single element and a single identity map, an object in $\mathcal{C}$ is a functor $\mathbb{1} \to \mathcal{C}$ and a map in $\mathcal{C}$ is a natural transformation between its domain's functor and codomain's functor:




The commutative aspect of categorical diagrams then becomes an equality of string diagrams.

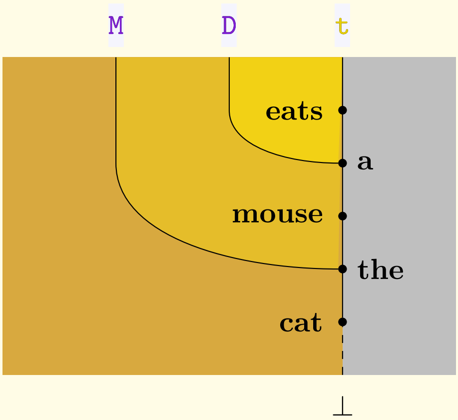
# Il était un petit garçon

By adding a string diagram rule for curryfication:



# Il était un petit garçon

We can compose string diagrams vertically to compute the meaning of full sentences:



# Penelope

Once we have the set of strings, we need to cut them and to see if different ways to cut them are equivalent by using equations:

# Penelope

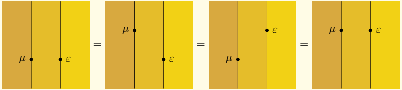
Once we have the set of strings, we need to cut them and to see if different ways to cut them are equivalent by using equations:

**Theorem 3.2 — Theorem 3.1 [?], Theorem 1.2 [?]** A well-formed equation between morphism terms in the language of monoidal categories follows from the axioms of monoidal categories if and only if it holds, up to planar isotopy, in the graphical language.

# Penelope

Let us now look at a few of the equations that arise from the commutation of certain diagrams:

The *Elevator* Equations are a consequence of Theorem 3.1

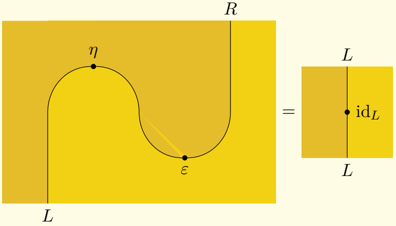



(🛗)

# Penelope

Let us now look at a few of the equations that arise from the commutation of certain diagrams:

**The *Snake* Equations** If we have an adjunction $L \dashv R$:

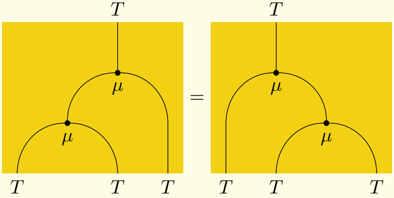



(🐍)

# Penelope

Let us now look at a few of the equations that arise from the commutation of certain diagrams:

## The *(co-)Monadic* Equations




$(\mu)$

# 1, 2, 3

Following [?] we represent a diagram $D$ as a triplet:

# 1, 2, 3

Following [?] we represent a diagram $D$ as a triplet:

- an integer $D.N$ for the height of $D$.

# 1, 2, 3

Following [?] we represent a diagram $D$ as a triplet:

- an integer $D.N$ for the height of $D$.
- an array $D.S$ for the types of the input strings.

# 1, 2, 3

Following [?] we represent a diagram $D$ as a triplet:

- an integer $D.N$ for the height of $D$.
- an array $D.S$ for the types of the input strings.
- a function $D.L$ which takes a natural number smaller than $D.N - 1$ and returns its type as a tuple of arrays $nat = (nat.\mathrm{h}, nat.\mathrm{in}, nat.\mathrm{out})$.

# 1, 2, 3

To check for validity we can in quasi-linear time:

**function** $\textsc{isValid}(D)$

$S \leftarrow D.\mathrm{S}$

**for** $i < D.\mathrm{N}$ **do**

$nat \leftarrow D.\mathrm{L}\,(i)$

$b, e \leftarrow nat.\mathrm{h}, nat.\mathrm{h} + |nat.\mathrm{in}|$

**if** $S[b : e] \neq nat.\mathrm{in}$ **then return** False

$S[b : e] \leftarrow nat.\mathrm{out}$ ▷ To understand as replacing in place.

**return** True

[**?**]'s algorithm for right reductions is still valid.

# Lyon

**Theorem 3.3 — Confluence** Our reduction system is confluent and therefore defines normal forms:

1. Right reductions are confluent and therefore define *right* normal forms for diagrams under the equivalence relation induced by exchange.
2. Equational reductions are confluent and therefore define *equational* normal forms for diagrams under the equivalence relation induced by exchange.

# Lyon

We can reduce a diagram $D$ in right normal form along (**??**) at height $i$ if, and only if:

- $D.\mathrm{L}\left(i\right).\mathrm{h} = D.\mathrm{L}\left(i+1\right).\mathrm{h} - 1$

# Lyon

We can reduce a diagram $D$ in right normal form along (**??**) at height $i$ if, and only if:

- $D.\mathrm{L}\left(i\right).\mathrm{h} = D.\mathrm{L}\left(i+1\right).\mathrm{h} - 1$
- $D.\mathrm{L}\left(i\right) = \eta_{L,R}$

# Lyon

We can reduce a diagram $D$ in right normal form along (**??**) at height $i$ if, and only if:

- $D.\mathrm{L}\left(i\right).\mathrm{h} = D.\mathrm{L}\left(i+1\right).\mathrm{h} - 1$
- $D.\mathrm{L}\left(i\right) = \eta_{L,R}$
- $D.\mathrm{L}\left(i+1\right) = \varepsilon_{L,R}$

# Lyon

We can reduce a diagram $D$ in right normal form along (**??**) at height $i$ if, and only if:

- $D.\mathrm{L}\left(i\right).\mathrm{h} = D.\mathrm{L}\left(i+1\right).\mathrm{h} - 1$
- $D.\mathrm{L}\left(i\right) = \eta_{L,R}$
- $D.\mathrm{L}\left(i+1\right) = \varepsilon_{L,R}$

The reduction is done by deleting $i$ and $i+1$ from $D$ and reindexing.

# Lyon

We can reduce a diagram $D$ in right normal form along (**??**) at height $i$ if, and only if:

- $D.\mathrm{L}\left(i\right).\mathrm{h} = D.\mathrm{L}\left(i+1\right).\mathrm{h} - 1$
- $D.\mathrm{L}\left(i\right) = \eta_{L,R}$
- $D.\mathrm{L}\left(i+1\right) = \varepsilon_{L,R}$

The reduction is done by deleting $i$ and $i+1$ from $D$ and reindexing.

We ask that equation **??** is always used in the direct sense $\mu\left(\mu\left(TT\right), T\right) \to \mu\left(T\mu\left(TT\right)\right)$ so that the algorithm terminates. We never should be needing Equation **??** in this part of the workflow.

# Lyon

We can reduce a diagram $D$ in right normal form along (**??**) at height $i$ if, and only if:

- $D.\mathrm{L}\left(i\right).\mathrm{h} = D.\mathrm{L}\left(i+1\right).\mathrm{h} - 1$
- $D.\mathrm{L}\left(i\right) = \eta_{L,R}$
- $D.\mathrm{L}\left(i+1\right) = \varepsilon_{L,R}$

The reduction is done by deleting $i$ and $i+1$ from $D$ and reindexing.

We ask that equation **??** is always used in the direct sense $\mu\left(\mu\left(TT\right), T\right) \to \mu\left(T\mu\left(TT\right)\right)$ so that the algorithm terminates. We never should be needing Equation **??** in this part of the workflow.

# Lyon

*Proof of the Confluence Theorem.* The first point of this theorem is exactly Theorem 4.2 in [**?**]. To prove the second part, note that the reduction process terminates as we strictly decrease the number of $2$-cells with each reduction. Moreover, our claim that the reduction process is confluent is obvious from the associativity of Equation (**??**) and the fact the other equations delete node and simply delete equations, completing our proof. ■

# Normale

**Theorem 3.4 — Normalization Complexity** Reducing a diagram to its normal form is done in quadratic time in the number of natural transformations in it.

# Plan

# CONTACT

- Do we need to adapt to other denotational processes ?

# CONTACT

- Do we need to adapt to other denotational processes ?
- How to quicken the parsing ?

# CONTACT

- Do we need to adapt to other denotational processes ?
- How to quicken the parsing ?
- How to reconcile $\mathtt{M}a \rightarrow b$ functions and DisCoCat [**?**] ?

# Conclusion

In this presentation:

## Conclusion

In this presentation:

- We showed how to define a categorical type semantics given a typed lexicon of a language.

# Conclusion

In this presentation:

- We showed how to define a categorical type semantics given a typed lexicon of a language.
- We showed how to introduce and compose effects in such a semantic, avoiding quantification difficulties in typing.

## Conclusion

In this presentation:

- We showed how to define a categorical type semantics given a typed lexicon of a language.
- We showed how to introduce and compose effects in such a semantic, avoiding quantification difficulties in typing.
- We proposed a way to introduce more general concepts as.
- We showed that it is possible to efficiently represent and reduce the effect-solving process, independently of the speaker.

# Bibliography I

| **Expression** | Type | $\lambda$-Term |
|---|---|---|
| **planet** | $\mathtt{e} \to \mathtt{t}$ | $\lambda x.\mathbf{planet}\, x$ |
| | Generalizes to **common nouns** | |
| **carnivorous** | $\left(\mathtt{e} \to \mathtt{t}\right)$ | $\lambda x.\mathbf{carnivorous}\, x$ |
| | Generalizes to **predicative adjectives** | |
| **skillful** | $\left(\mathtt{e} \to \mathtt{t}\right) \to \left(\mathtt{e} \to \mathtt{t}\right)$ | $\lambda p.\lambda x.px \wedge \mathbf{skillful}\, x$ |
| | Generalizes to **predicate modifier adjectives** | |
| **Jupiter** | $\mathtt{e}$ | $\mathbf{j} \in \mathrm{Var}$ |
| | Generalizes to **proper nouns** | |
| **sleep** | $\mathtt{e} \to \mathtt{t}$ | $\lambda x.\mathbf{sleep}\, x$ |
| | Generalizes to **intranitive verbs** | |
| **chase** | $\mathtt{e} \to \mathtt{e} \to \mathtt{t}$ | $\lambda o.\lambda s.\mathbf{chase}\,(o)\,(s)$ |
| | Generalizes to **transitive verbs** | |
| **be** | $\left(\mathtt{e} \to \mathtt{t}\right) \to \mathtt{e} \to \mathtt{t}$ | $\lambda p.\lambda x.px$ |

| Constructor | fmap | Typeclass |
|---|---|---|
| $\mathtt{G}\,(\tau) = \mathtt{r} \to \tau$ | $\mathtt{G}\varphi\,(x) = \lambda r.\varphi\,(xr)$ | Monad |
| $\mathtt{W}\,(\tau) = \tau \times \mathtt{t}$ | $\mathtt{W}\varphi\,(\langle a, p\rangle) = \langle \varphi a, p\rangle$ | Monad |
| $\mathtt{S}\,(\tau) = \{\tau\}$ | $\mathtt{S}\varphi\,(\{x\}) = \{\varphi(x)\}$ | Monad |
| $\mathtt{C}\,(\tau) = \left(\tau \to \mathtt{t}\right) \to \mathtt{t}$ | $\mathtt{C}\varphi\,(x) = \lambda c.x\,(\lambda a.c\,(\varphi a))$ | Monad |
| $\mathtt{T}\,(\tau) = \mathtt{s} \to \left(\tau \times \mathtt{s}\right)$ | $\mathtt{D}\varphi\,(\lambda s.\,\{\langle x, x + s\rangle \mid px\}) = \lambda s.\,\langle \varphi x, \varphi x + s\rangle$ | Monad |
| $\mathtt{F}\varphi\,(\tau) = \tau \times \mathtt{S}\tau$ | $\mathtt{F}\,(\langle v, \{x \mid x \in D_e\}\rangle) = \langle \varphi\,(v)\,, \{x \mid x \in D_e\}\rangle$ | Monad |
| $\mathtt{D}\,(\tau) = \mathtt{s} \to \mathtt{S}\left(\mathtt{e} \times \mathtt{s}\right)$ | $\mathtt{D}\varphi\,(\lambda s.\,\{\langle x, x + s\rangle \mid px\}) = \lambda s.\,\{\langle \varphi x, \varphi x + s\rangle \mid px\}$ | Monad |
| $\mathtt{M}\,(\tau) = \tau + \perp$ | $\mathtt{M}\varphi\,(x) = \begin{cases} \varphi\,(x) & \text{if } \Gamma \vdash x : \tau \\ \# & \text{if } \Gamma \vdash x : \# \end{cases}$ | Monad |

| | | |
|---|---|---|
| **CN(P)** | $\Gamma \vdash p : (\mathtt{e} \to \mathtt{t})$ | $\Pi(p) = \lambda x.\,(px \wedge \lvert x \rvert \geq 2)$ |
| **ADJ(P)** | $\Gamma \vdash p : (\mathtt{e} \to \mathtt{t})$ | $\Pi(p) = \lambda x.\,(px \wedge \lvert x \rvert \geq 2)$ |
| | $\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t})$ | $\Pi(p) = \lambda \nu.\lambda x.\,(p\,(\nu)\,(x) \wedge \lvert x \rvert \geq 2)$ |
| **NP** | $\Gamma \vdash p : \mathtt{e}$ | $\Pi(p) = p$ |
| | $\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$ | $\Pi(p) = \lambda \nu.p\,(\Pi \nu)$ |
| **IV(P)/VP** | $\Gamma \vdash p : \mathtt{e} \to \mathtt{t}$ | $\Pi(p) = \lambda o.\,(po \wedge \lvert x \rvert \geq 2)$ |
| **TV(P)** | $\Gamma \vdash p : \mathtt{e} \to \mathtt{e} \to \mathtt{t}$ | $\Pi(p) = \lambda s.\lambda o.\,(p\,(s)\,(o) \wedge \lvert s \rvert \geq 2)$ |
| **REL(P)** | $\Gamma \vdash p : \mathtt{e} \to \mathtt{t}$ | $\Pi(p) = \lambda x.\,(px \wedge \lvert x \rvert \geq 2)$ |
| **DET** | $\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to ((\mathtt{e} \to \mathtt{t}) \to \mathtt{t})$ | $\Pi(p) = \lambda \nu.p\,(\Pi \nu)$ |
| | $\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to \mathtt{e}$ | $\Pi(p) = \lambda \nu.p\,(\Pi \nu)$ |