

On a Categorical Type and Effect Inference Structure for Semantic Denotation Combinations in Natural Languages: A Purely Functional Analysis of English

Matthieu Boyer

May 2, 2025



Contents

1	Categorical Semantics of Effects: A Typing System	2
1.1	Typing Category	3
1.1.1	Types	3
1.1.2	Functors, Applicatives and Monads	3
1.1.3	Natural Transformations for Handlers and Higher-Order Constructs	4
1.2	Typing Judgements	6
2	Handling Non-Determinism	7
2.1	String Diagram Modelisation of Sentences	7
2.2	Achieving Normal Forms	9
2.2.1	The Composition Dilemma	9
2.2.2	Equations	9
2.3	Computing Normal Forms	10
2.3.1	Representing String Diagrams	11
2.3.2	Equational Reductions	11
3	Semantic Parsing	13
3.1	Naïve Semantic Parsing on Syntactic Parsing	13
3.1.1	Islands	15
3.2	Rewriting Rules	15
4	Language Translation	16
4.1	Syntax	16
4.2	Lambda-Calculus	16
4.2.1	Types of Syntax	16

4.2.2	Lexicon: Semantic Denotations for Words	16
4.2.3	Effects of the Language	16
4.2.4	Modality, Plurality, Aspect, Tense	17
4.3	Other Representation of Concepts	18
4.3.1	Manifold Representation	18
4.3.2	Vector Representation	18
4.3.3	Probabilistic Monad	18
A	Other Considered Things	19
A.1	Typing with a Product Category (and a bit of polymorphism)	19
A.2	Enhanced Grammar	19

Abstract

The main idea is that you can consider some words to act as functors in a typing category, for example determiners: they don't change the way a word acts in a sentence, but more the setting in which the word works by adding an effect to the work. The linguistics is based mostly on work by ?.

Introduction

? provided a way to view monads as effects in functional programming. This allows for new modes of combination in a compositional semantic formalism, and provides a way to model words which usually raise problems with the traditional lambda-calculus representation of the words. In particular we consider words such as *the* or *a* whose application to common nouns results in types that should be used in similar situations but with largely different semantics, and model those as functions whose application yields an effect. This allows us to develop typing judgements and an extended typing system for compositional semantics of natural languages. Type-driven compositional semantics acts under the premise that given a set of words and their denotations, a set of grammar rules for composition and their associated typing judgements, we are able to form an enhanced parser for natural language which provides a mathematical representation of the meaning of sentences, as proposed by ?.

This is not the first time a categorical representation of a compositional semantics of natural language is proposed, ? already suggested an approach based on monoidal categories using an outside model of meaning. However, what we propose here is a representation of the different capabilities of words as categorical constructs: we allow for a wider set of representations inside our model of meaning, trading non-determinism for additional structures. In this regard, we basically combine the grammatical type and the meaning of a word by having our denotations be associated with a type: there is no need from an additional category outside of our typing category and we limit ourselves to reducing non-determinism by limiting our possibilities for combinations with a provided CFG¹ of the language.

The focus of our system is to allow more flexibility in denotations, leading to more possibilities of combinations.

1 Categorical Semantics of Effects: A Typing System

In this section, we will designate by \mathcal{L} our language, as a set of words (with their semantics) and syntactic rules to combine them semantically. We denote by $\mathcal{O}(\mathcal{L})$ the set of words in the language whose semantic representation is a low-order function and $\mathcal{F}(\mathcal{L})$ the set of words whose semantic representation is a functor or high-order function. Our goals here are to describe more formally, using a categorical vocabulary, the environment in which the typing system for our language will exist, and how we connect words and other linguistic objects to the categorical formulation.

¹Or another model that could generate our language.

```

0, 0, Hello,
1, 10, Hello,
2, 20, Hello,
3, 30, Hello,
4, 40, Hello,
5, 50, Hello,
6, 60, Hello,
7, 70, Hello,
8, 80, Hello,
9, 90, Hello,
10, 100, Hello,
F F F e→t

```

1.1 Typing Category

1.1.1 Types

Let \mathcal{C} be a closed cartesian category. This represents our main typing system, consisting of words $\mathcal{O}(\mathcal{L})$ that can be expressed without effects. Remember that \mathcal{C} contains a terminal object \perp representing the empty type or the lack thereof. We can consider $\bar{\mathcal{C}}$ the category closure of $\mathcal{F}(\mathcal{L}) (\mathcal{O}(\mathcal{L}))$, that is consisting of all the different type constructors (ergo, functors) that could be formed in the language. What this means is that we consider for our category objects any object that can be attained in a finite number from a finite number of functorial applications from an object of \mathcal{C} . In that sense, $\bar{\mathcal{C}}$ is still a closed cartesian category (since all our functors induce isomorphisms on their image)². $\bar{\mathcal{C}}$ will be our typing category (in a way).

We consider for our types the quotient set $\star = \text{Obj}(\bar{\mathcal{C}}) / \mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ does not induce an equivalence relation on $\text{Obj}(\bar{\mathcal{C}})$ but a preorder, we consider the chains obtained by the closure of the relation $x \succeq y \Leftrightarrow \exists F, y = F(x)$ (which is seen as a subtyping relation as proposed in ?). We also define \star_0 to be the set obtained when considering types which have not yet been *affected*, that is $\text{Obj}(\mathcal{C})$. In contexts of polymorphism, we identify \star_0 to the adequate subset of \star . In this paradigm, constant objects (or results of fully done computations) are functions with type $\perp \rightarrow \tau$ which we will denote directly by $\tau \in \star_0$.

What this construct actually means, in categorical terms, is that a type is an object of $\bar{\mathcal{C}}$ (as intended) but we add a subtyping relationship based on the procedure used to construct $\bar{\mathcal{C}}$. Note that we can translate that subtyping relationship on functions as $F \left(A \xrightarrow{\varphi} B \right)$ has types $F(A \Rightarrow B)$ and $FA \Rightarrow FB$.

We will provide in Table 2 a list of the effect-less usual types associated to regular linguistic objects.

1.1.2 Functors, Applicatives and Monads

Our point of view leads us to consider *language functors*³ as polymorphic functions: for a (possibly restrained, though it seems to always be \star) set of base types S , a functor is a function

$$x : \tau \in S \subseteq \star \mapsto Fx : F\tau$$

Remember that \star is a fibration of the types in $\bar{\mathcal{C}}$. This means that if a functor can be applied to a type, it can also be applied to all *affected* versions of that type, i.e. $\mathcal{F}(L)(\tau \in \star)$. More importantly, while it seems that F 's type is the identity on \star , the important part is that it changes the effects applied to x (or τ). In that sense, F has the following

²Our definition does not yield a closed cartesian category as there are no exponential objects between results of two different functors, but this is not a real issue as we can just say we add those.

³The elements of our language, not the categorical construct.

typing judgements:

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

We use the same notation for the *language functor* and the *type functor* in the examples, but it is important to note those are two different objects, although connected. More precisely, the *language functor* is to be seen as a function whose computation yields an effect, while the *type functor* is the endofunctor of $\bar{\mathcal{C}}$ (so a functor from \mathcal{C}) that represents the effect in our typing category.

In the same fashion, we can consider functions to have a type in \star or more precisely of the form $\star \rightarrow \star$ which is a subset of \star . This justifies how functors can act on functions in our typing system, thanks to the subtyping judgement introduced above, as this provides a way to ensure proper typing while just propagating the effects. Because of propagation, this also means we can resolve the effects or keep on with the computation at any point during parsing, without any fear that the results may differ.

In that sense, applicatives and monads only provide with more flexibility on the ways to combine functions: they provide intermediate judgements to help with the combination of trees. For example, the multiplication of the monad provides a new *type conversion* judgement:

$$\frac{\Gamma \vdash x : MM\tau}{\Gamma \vdash x : M\tau \succeq MM\tau} \text{Monad}$$

This is actually a special case of the natural transformation rule that we define below, which means that, in a way, types $MM\star$ and $M\star$ are equivalent, as there is a canonical way to go from one type to another. Remember however that $M\star$ is still a proper subtype of $MM\star$ and that the objects are not actually equal: they are simply equivalent.

1.1.3 Natural Transformations for Handlers and Higher-Order Constructs

We could also add judgements for adjunctions, but the most interesting thing is to add judgements for natural transformations, as adjunctions are particular examples of natural transformations which arise from *natural* settings. While in general we do not want to find natural transformations, we want to be able to express these in three situations:

1. If we have an adjunction $L \dashv R$, we have natural transformations for $\text{Id}_{\mathcal{C}} \Rightarrow L \circ R$ and $R \circ L \Rightarrow \text{Id}_{\mathcal{C}}$. In particular we get a monad and a comonad from a canonical setting.
2. To deal with the resolution of effects, we can map handlers to natural transformations which go from some functor F to the Id functor, allowing for a sequential⁴ computation of the effects added to the meaning. We will develop a bit more on this idea in Paragraph 1.1.3.2 and in Section 2.
3. To create *higher-order* constructs which transform words from our language into other words, while keeping the functorial aspect. This idea is developed in 1.1.3.3.

To see why we want this rule, which is a larger version of the monad multiplication and the monad/applicative unit, it suffices to see that the diagram below provides a way to construct the “correct function” on the “correct functor” side of types. If we have a natural transformation $F \xRightarrow{\theta} G$ then for all arrows $f : \tau_1 \rightarrow \tau_2$ we have:

$$\begin{array}{ccc} F\tau_1 & \xrightarrow{Ff} & F\tau_2 \\ \theta_{\tau_1} \downarrow & & \downarrow \theta_{\tau_2} \\ G\tau_1 & \xrightarrow{Gf} & G\tau_2 \end{array}$$

and this implies, from it being true for all arrows, that from $\Gamma \vdash x : F\tau$ we have an easy construct to show $\Gamma \vdash x : G\tau$.

Remember that in the Haskell programming language, any polymorphic function is a natural transformation from the first type constructor to the second type constructor, as proved by ?. This will guarantee for us that given a *Haskell* construction for a polymorphic function, we will get the associated natural transformation.

⁴In particular, non-necessarily commutative

1.1.3.1 Adjunctions

We will not go in much details about adjunctions, as a full example and generalization process is provided in 4.2.3. First, we remind the definition: an adjunction $L \dashv R$ is a pair of functors $L : \mathcal{A} \rightarrow \mathcal{B}$ and $R : \mathcal{B} \rightarrow \mathcal{A}$, and a pair of natural transformations $\eta : \text{Id}_{\mathcal{A}} \Rightarrow R \circ L$ and $\varepsilon : L \circ R \Rightarrow \text{Id}_{\mathcal{B}}$ such that the two following equations are satisfied:

A TikZ PICTURE GOES HERE.



A TikZ PICTURE GOES HERE.



An adjunction defines two different structures over itself: a monad $L \circ R$ and a comonad $R \circ L$. The fact these structures arise from the interaction between two effects renders them an intrinsic property of the language. In this lies the usefulness of adjunction in a typing system which uses effects: adjunctions provide a way to combine effects and to handle effects, allowing to simplify the computations on the free monoid on the set of functors.

1.1.3.2 Handlers

As introduced by ?, the use of handlers as annotations to the syntactic tree of the sentence is an appropriate formalism. This could also give us a way to construct handlers for our effects as per ?, or ?. As considered by ? and ?, handlers are to be seen as natural transformations describing the free monad on an algebraic effect. Considering handlers as so, allows us to directly handle our computations inside our typing system, by “transporting” our functors one order higher up without loss of information or generality since all our functors undergo the same transformation. Using the framework proposed in (?) we simply need to create handlers for our effects/functors and we will then have in our language the result needed. The only thing we will require from an algebraic handler h is that for any applicative functor of unit η , $h \circ \eta = \text{id}$.

What does this mean in our typing category ? It means that either our language or our parser for the language ⁵ should contain natural transformations $F \Rightarrow \text{Id}$ for $F \in \mathcal{F}(\mathcal{L})$. In this goal, we remember that from any polymorphic function in *Haskell* we get a natural transformation (?) meaning that it is enough to be able to define our handler in *Haskell* to be ensured of its good definition. Note that the choice of the handler being part of the lexicon or the parser over the other is a philosophical question more than a semantical one, as both options will result in semantically equivalent models, the only difference will be in the way we consider the resolution of effects. Mathematically⁶, this means the choice of either one of the options is purely of detail left during the implementation.

However, this choice does not arise in the case of the adjunction-induced handlers. Indeed here, the choice is caused by the non-uniqueness of the choices for the handlers. For example, two different speakers may have different ways to resolve the **S** (which will be introduced as the *Powerset* monad in Section 4.2.3) that arises from the phrase *A chair*. This usual example of the differences between the cognitive representation of words is actually a specific example of the different possible handlers for the powerset representation of non-determinism/indefinites: there are $|S|$ arrows from the initial object to S in *Set*, representing the different elements of S . In that sense, while handlers may have a normal form or representation purely dependant on the effect, the actual handler does not necessarily have a canonical form. This is the difference with the adjunctions: adjunctions are intrinsic properties of the coexistence of the effects⁷, while the handlers are user-defined. As such, we choose to say that our handlers are implemented parser-side but again, this does not change our modelisation of handlers as natural transformations and most importantly, this does not add non-determinism to our model: The non-determinism that arises from the variety of possible handlers does not add to the non-determinism in the parsing.

⁵Depending whether we think handling effects is an intrinsic construct of the semantics of the language or whether it is associated with a speaker.

⁶Computer-wise, actually.

⁷Which we identify to their functorial representations, which we may identify to their free monad in the framework (?).

A TikZ PICTURE GOES HERE.

Table 1: Typing and Subtyping Judgements

1.1.3.3 Higher-Order Constructs

We might want to add plurals, superlatives, tenses, aspects and other similar constructs which act as function modifiers. For each of these, we give a functor Π corresponding to a new class of types along with natural transformations for all other functors F which allows to propagate down the high-order effect. This transformation will need to be from $\Pi \circ F$ to $\Pi \circ F \circ \Pi$ or simply $\Pi \circ F \Rightarrow F \circ \Pi$ depending on the situation. This allows us to add complexity not in the compositional aspects but in the lexicon aspects. We do not want these functors to be applicatives, as we do not want a way to *create* them in the parser if they are not marked in the sentence.

One of the main issues with this is the following: In the English language, plural is marked on all words (except verbs, and even then case could be made for it to be marked), while future is marked only on verbs (through the *will + verb* construct which creates a “new” verb in a sense) though it applies also to the arguments on the verb. A way to solve this would be to include in the natural transformations rules to propagate the functor depending on the type of the object. Consider the superlative effect **most**⁸. As it can only be applied on adjectives, we can assume its argument is a function (but the definition would hold anyway taking $\tau_1 = \perp$). It is associated with the following function (which is a rewriting of the natural transformation rule):

$$\frac{\Gamma \vdash x : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathbf{most} x := \Pi_{\tau_2} \circ x = x \circ \Pi_{\tau_1}}$$

What curryfication implies, is that higher-order constructs can be passed down to the arguments of the functions they are applied to, explaining how we can reconcile the following semantic equation even if some of the words are not marked properly:

$$\mathbf{future}(\mathbf{be}(\mathbf{I}, \mathbf{a\ cat})) = \mathbf{will\ be}(\mathbf{future}(\mathbf{I}), \mathbf{a\ cat}) = \mathbf{be}(\mathbf{future}(\mathbf{I}), \mathbf{a\ cat})$$

Indeed the above equation could be simply written by our natural transformation rule as:

$$\mathbf{future}(\mathbf{be})(\mathbf{arg}_1, \mathbf{arg}_2) = \mathbf{future}(\mathbf{be})(\mathbf{arg}_2)(\mathbf{future}(\mathbf{arg}_1)) = \mathbf{future}(\mathbf{be})(\mathbf{future}(\mathbf{arg}_2))(\mathbf{future}(\mathbf{arg}_1))$$

This is not a definitive rule as we could want to stop at a step in the derivation, depending on our understanding of the notion of future in the language.

1.2 Typing Judgements

To complete this section, Table ?? gives a simple list of different typing composition judgements through which we also re-derive the subtyping judgement to allow for its implementation. Note that here, the syntax is not taken into account: a function is always written left of its arguments, whether or not they are actually in that order in the sentence. This issue will be resolved by giving the syntactic tree of the sentence (or inferring it at runtime). We could also add symmetry induced rules for application.

Furthermore, note that the App rule is the **fmap** rule for the identity functor and that the **pure/return** and **>>=** is the **nat** rule for the monad unit (or applicative unit) and multiplication.

Using these typing rules for our combinators, it is important to see that our grammar will still be ambiguous and thus our reduction process will be non-deterministic. As an example, we provide the typing reductions for the classic

⁸We do not care about morphological markings here, we say that **largest** = **most** (**large**)

A TikZ PICTURE GOES HERE.

Figure 1: Parsing trees for the typing of *The man sees the girl using a telescope*.

example: *The man sees the girl using a telescope* in Figure 1.

This non-determinism is a component of our language's grammar and semantics: a same sentence can have multiple interpretation without context. By the same reasoning we applied on handlers, we choose not to resolve the ambiguity in the parser but we will try in Section 2 to reduce it to a minimum. Even so some of our derivations will include ways to read from a context (for example the **Jupiter**, a **planet** construct), we forget about the definition of the context, or its updating after a sentence, in our first proposition.

2 Handling Non-Determinism

The typing judgements proposed in Section 1.2 lead to ambiguity. In this section we propose ways to get our derivations to a certain normal form, by deriving an equivalence relation on our derivation and parsing trees, based on string diagrams.

2.1 String Diagram Modelisation of Sentences

String diagrams are the Poincaré duals of the usual categorical diagrams when considered in the 2-category of categories and functors. In this category, the endofunctors are natural transformations. This means that we represent categories as regions of the plane, functors as lines separating regions and natural transformations as the intersection points between two lines. We will change the color of the regions when crossing a functor, and draw the identity functor as a dashed line or no line at all in the following sections. We will always consider application as applying to the right of the line so that composition is written in the same way as in equations:

A TikZ PICTURE GOES HERE.

This gives us a new graphical formalism to represent our effects using a few equality rules between diagrams. The commutative aspect of functional diagrams is now replaced by an equality of string diagrams, which will be detailed in the following section.

The important aspect of string diagrams that we will use is that two diagrams that are planarily homotopic are equal (?) and that we can map a string diagram to a sequence of computations on computation: the vertical composition of natural transformations (bottom-up) represents the reductions that we can do on our set of effects. This means that even when adding handlers, we get a way to visually see the meaning get reduced from effectful composition to propositional values, without the need to specify what the handler does. Indeed we only look at *when* we apply handlers and natural transformations reducing the number of effects. This delimits our usage of string diagrams as ways to look at computations and a tool to provide equality rules to reduce non-determinism by constructing an equivalence relationship (which we denote by \equiv) and yielding a quotient set of *normal forms* for our computations.

First let's look at the fact we can see functors as natural transformations and morphisms as functors, to justify that our typing tree are, in a sense string diagrams. Let us define the category $\mathbb{1}$ with exactly one object and one arrow: the identity on that object. It will be shown in grey in the string diagrams below. A functor of type $\mathbb{1} \rightarrow \mathcal{C}$ is equivalent to choosing an object in \mathcal{C} , and a natural between two such functors τ_1, τ_2 is exactly an arrow in \mathcal{C} of type $\tau_1 \rightarrow \tau_2$. This gives us a way to map the composition of our sentence to a simple string in our diagram. Knowing that allows us to represent the type resulting from a sequence of computations as a sequence of strings whose farthest right represents an object in \mathcal{C} , that is, a base type.

A TikZ PICTURE GOES HERE.

For simplicity reasons, and because the effects that are buried in our typing system not only give rise to functors but also have types that are not purely curriable, we will write our string diagrams on the fully parsed sentence, with its most simplified/composed expression. Indeed, the question of providing rules to compose the string diagram for **the** and the one for **cat** to give the one for **the cat** is a difficult question, that natural solutions to are not obvious⁹. The natural composition rules on diagrams being the gluing together of diagrams, we do need to add something to our toolbox.

A TikZ PICTURE GOES HERE. (CSD)

To justify our proposition to only consider fully reduced expressions note that in this formalism we don't consider the expressions for our functors and natural transformations¹⁰ but simply the sequence in which they are applied. In particular, this means the following diagrams commute for any F, G functors and θ natural transformation.

$$\begin{array}{ccc}
 G & \xrightarrow{F} & F \circ G \\
 \theta \downarrow & & \downarrow F \circ \theta \\
 \theta G & \xrightarrow{F} & F \circ \theta G
 \end{array}
 \qquad
 \begin{array}{ccc}
 G & \xrightarrow{F} & G \circ F \\
 \theta \downarrow & & \downarrow \theta \circ F \\
 \theta G & \xrightarrow{F} & \theta G \circ F
 \end{array}$$

The property of natural transformations to be applied before or after any arrow in the category justifies that even when composing before the handling we get the same result. These properties justifies the fact that we only need to prove the equality of two reductions at the farthest step of the reductions, even though in practice the handling might be done at earlier points in the parsing. We have indeed covered all the composition options of our language by talking about functors and arrows.

In the end, we will have the need to go from a certain set of strings (the effects that applied) to a single one, through a sequence of handlers, monadic and comonadic rules and so on. Notice that we never reference the zero-cells and that in particular their colors are purely an artistical touch¹¹. In the following sections, we will differentiate the different *regions* of $\bar{\mathcal{C}}$ by changing colours each time we cross a functor, which is actually an endofunctor in $\bar{\mathcal{C}}$ and thus would not imply a colour change, as can be seen in Equations (η) and (μ) .

A TikZ PICTURE GOES HERE.

It is important to acknowledge that a previous modelisation of parsers by string diagrams has already been proposed (?. ?). When considering their way of using string diagrams for syntactic reductions, we can use our effects to improve the typing by asking that effects arising from a computation are sent to an axis orthogonal to the one used to represent the sentence, allowing for a reduction on a second axis in a second time.

Since we provide a compositional addition to a denotation system, it amounts to replacing/improving the category of meaning in the product category with the pregroup category and thus means we can integrate our theory inside their formalism. Indeed, we start from a meaning category \mathcal{C} and then build on it, instead of providing a full on denotation system¹². Moreover, when seeing their parsing trees as string diagrams and using the fact that we can always assume natural transformations (or 2-cells) to be at different heights¹³, we can see that their model of string diagrams is just a

⁹I will suggest a solution when discussing the set of equations between our diagrams and the way we construct our *normal forms*.

¹⁰Here, we are talking about the actual *natural transformations* that are used for handling effects, whether they're the ones associated to algebraic handlers or adjunctions/monads...

¹¹It looks nice and makes you forget this is applied abstract nonsense.

¹²Section 4 proposes a translation of the english language based on usual denotations, but our system could be applied onto a fully different representation of meaning, as long as it can be mapped to a cartesian closed category.

¹³Geometrically, see Lemma 1.7 in ?.

rewriting of our model for vertical composition, and allows for other reductions in the parser to reduce non-determinism. We keep the work on vertically composing string diagrams as a way to see how the effects pile up, as presented in Section 2.2.1.

There is however one thing that may seem to be an issue: existential quantifiers inside **if then** sentences and other functions looking like $\mathbf{M}a \rightarrow b$. Indeed, (?) suggests that those could be of type $\mathbf{S}t \rightarrow t$ and that we might want to apply that function to something of type $\mathbf{S}t$ by first using the unit of the monad \mathbf{S} then invoking \mathbf{fmap} . There is no issue with that in our typing system, as this combination mode is the composition of two typing judgements (in a sense). To graphically reconcile this and our string diagrams, especially since units and handlers cancel each other, we suggest to see the effects as being dragged along the parsing trees (or parsing string diagrams) until it's never needed again. This means that edges in parsing trees (or strings in string diagrams) could be viewed as bundles of effects with individual strings being sent as proposed before on an axis in the order in which they arrive once no function (or composition mode) requires its presence. As such, we forget that units actually appear in the composition modes.

2.2 Achieving Normal Forms

We will now provide a set of rewriting rules on string diagrams (written as equations) which define the set of different possible reductions and explain how our typing rules compose diagrammatically.

2.2.1 The Composition Dilemma

In Equation (CSD) we provide an example of why composing string diagrams, in our paradigm, is not as simple as we would like. However, we could solve this conundrum by adding to our set of equations the following rule, which we denote as the *Curryfication* rule:

A TikZ PICTURE GOES HERE. (Curryfication)

This rule is to be seen as a straight-forward consequence of the isomorphism between $\mathcal{C}(A, B)$ the arrows between A and B and $A \Rightarrow B$ the exponential object of B to the A . Allowing for that and remembering the fact that you can always replace a diagram by an equivalent one before composing answers the question of how we create composition rules for string diagrams, or almost: there is still a situation that is not accounted for, or at least explained: what happens when applying an effectful diagram¹⁴ to another effectful diagram? Take the equation below:

A TikZ PICTURE GOES HERE.

Note that the effects are applied in the order of the application, meaning that if we apply a function $\varphi : \tau_1 \rightarrow \tau_2 \rightarrow \tau_3$ to an element of type $F\tau_1$ then to an element of type $F'\tau_2$ the result is of type $F'F\tau_3$ while applying in the other way around leads to a result of type $FF'\tau_3$ (by considering the lambdas interexchangeable).

2.2.2 Equations

First, Theorem 2.1 reminds the main result by ? about string diagrams which shows that our artistic representation of diagrams is correct and does not modify the equation or the rule we are presenting.

Theorem 2.1 — **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.



¹⁴When the associated effect is monadic this type of diagram represents an arrow in the Kleisli category of the monad.

We will not prove this theorem here as it implies huge portions of graph theory and algebra which are out of the scope of this paper.

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 2.1 but also highlight one of the most important properties of string diagrams in their modelisation of multi-threaded computations: what happens on one string does not influence what happens on another in the same time:

A TikZ PICTURE GOES HERE. 

The *Snake* Equations are a rewriting of the categorical diagrams equations  and  of Paragraph 1.1.3.1 which are the defining properties of an adjunction. If we have an adjunction $L \dashv R$:

A TikZ PICTURE GOES HERE. 

A TikZ PICTURE GOES HERE. 

Note that we could express these equations using the following section, though it is, for now, a bit easier to keep it that way.

What the reduction means is that we have a commutative mapping:

$$\begin{array}{ccc} (\text{cloud with lightning}) & \xrightarrow{\text{string}} & (\text{two snakes}) \\ \downarrow \text{\texttt{\textbackslash reflectbox}} & & \downarrow \text{\texttt{\textbackslash reflectbox}} \\ (\text{cloud with lightning}) & \xrightarrow{\text{string}} & (\text{two snakes}) \end{array}$$

The *(co-)Monadic* Equations explain the very familiar sentence that a monad is just¹⁵ a monoid on the category of the endofunctors, that is, a multiplication law and a unit. Here we do not present the co-monadic equations which are the exact same one but with a few letter changes and the up and down reversed.

A TikZ PICTURE GOES HERE. (η)

A TikZ PICTURE GOES HERE. (μ)

This set of equations, when added to our composition rules from Section 2.2.1 explain all the different reductions that can be made to limit non-determinism in our parsing strategies. Indeed, considering the equivalence relation \mathcal{R} freely generated from $\left\{ (\text{Elevator}), (\text{Snake 1}), (\text{Snake 2}), (\mu), (\eta) \right\}$ and the equivalence relationship \mathcal{R}' of planar isotopy from Theorem 2.1, we get a set of normal forms \mathcal{N} from the set of all well-formed parsing diagrams \mathcal{D} defined by:

$$\mathcal{N} = (\mathcal{D}/\mathcal{R})/\mathcal{R}'$$

2.3 Computing Normal Forms

Now that we have a set of rules telling us what we can and cannot do in our model while preserving the equality of the diagrams, we provide a combinatorial description of our diagrams to help compute the possible equalities between multiple reductions of a sentence meaning.

¹⁵Which is a monad

? proposed a combinatorial description to check in linear time for equality under Theorem 2.1. However, this model does not suffice to account for all of our equations, especially as labelling will influence the equations for monads, comonads and adjunctions. To provide with more flexibility (in exchange for a bit of time complexity) we use the description provided and change the description of inputs and outputs of each 2-cell by adding types and enforcing types. In this section we formally describe the data structure we propose, as well as algorithms for validity of diagrams and a system of rewriting that allows us to compute the normal forms for our system of effects.

2.3.1 Representing String Diagrams

We follow ? in their combinatorial description of string diagrams. We describe a diagram by an ordered set of its 2-cells (the natural transformations) along with the number of input strings, for each 2-cell the following information:

- Its horizontal position: the number of strings that are right of it (we adopt this convention to match our graphical representation of effects: the number of strings is the distance to the base type).
- Its type: an array of effects (read from left to right) that are the inputs to the natural transformation and an array of effects that are the outputs to the natural transformation. The empty array represents the identity functor. Of course, we will not actually copy arrays and arrays inside our datastructure but simply copy labels which are keys to a dictionary containing such arrays to limit the size of our structure, allowing for $\mathcal{O}(1)$ access to the associated properties.

We will then write a diagram D as a tuple of 3 elements¹⁶: $(D.N, D.S, D.L)$ where $D.N$ is a positive integers representing the height (or number of nodes) of D , $D.S$ is an array for the input strings of D and where $D.L$ is a function which takes a natural number smaller than $D.N - 1$ and returns its type as a tuple of arrays $nat = (nat.h, nat.in, nat.out)$. From this, we propose a naive algorithm to check if a string diagram is valid or not: Note that here the algorithm does

Algorithm 1 Validity Check

```

function ISVALID( $D$ )
   $S \leftarrow D.S$ 
  for  $i < D.N$  do
     $nat \leftarrow D.L(i)$ 
     $b, e \leftarrow nat.h, nat.h + |nat.in|$ 
    if  $S[b : e] \neq nat.in$  then return False
     $S[b : e] \leftarrow nat.out$ 
  return True

```

▷ To understand as replacing in place.

not require that all effects are handled for validity.

Since our representation contains strictly more information (without slowing access by a non-constant factor) than the one it is based on, our datastructure supports the linear and polynomial time algorithms proposed with the structure by ?. This in particular means that our structure can be normalized in polynomial time to check for equality under Theorem 2.1. More precisely, the complexity of our algorithm is in $\mathcal{O}(n \times \sup_i |D.L(i).in| + |D.L(i).out|)$, which depends on our lexicon but most of the times will be linear time.

2.3.2 Equational Reductions




We are faced a problem when computing reductions using the equations for our diagrams which is that by definition, an equation is symmetric. To solve this issue, we only use equations from left to right to reduce as much as possible our result instead. This also means that trivially our reduction system computes normal forms: it suffices to re-apply the algorithm for recumbent equivalence after the rest of equational reduction is done. Moreover, note that all our



¹⁶We could write it as a tuple of 5 elements by replacing $D.L$ by three functions that lower level

reductions are either incompatible or commutative, which leads to a confluent reduction system, and the well definition of our normal forms:


Theorem 2.2 — 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.

Before proving the theorem, let us first provide the reductions for the different equations for our description of string diagrams. It suffices to provide the reductions for Equations , , μ and η as Equation  is a reformulation of Theorem 2.1 which is taken care of by the first point of our theorem.

The Snake Equations First, let's see when we can apply Equation  to a diagram D which is in *right* normal form, meaning it's been right reduced as much as possible. Suppose we have an adjunction $L \dashv R$. Then we can reduce D along () at i if, and only if:

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

This comes from the fact that we can't send either ε above η (or the other way around) using right reductions and that there cannot be any natural transformations between the two. Obviously Equation  has almost the same definition. Then, the reduction is easy: we simply delete both strings, removing i and $i+1$ from D and reindexing the other nodes.

The Monadic Equations For the monadic equations, we only use Equation η as a way to reduce the number of natural transformations, since the goal here is to attain normal forms and not find all possible reductions. We ask that equation μ is always used in the direct sense $\mu(\mu(TT), T) \rightarrow \mu(T\mu(TT))$ so that the algorithm terminates. We use the same convention for the comonadic equations. The validity conditions are as easy to define for the monadic equations as for the *snake* equations when considering diagrams in *right* normal forms. Then, for Equation (η) we simply delete the nodes and for Equation (μ) we switch the horizontal positions for i and $i+1$.

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 (μ) and the fact the other equations delete node and simply delete equations, completing our proof of Theorem 2.2. ■

Theorem 2.3 — Normalization Complexity Reducing a diagram to its normal form is done in quadratic time in the number of natural transformations in it.

Proof. Let's now give an upper bound on the number of reductions. Since each reductions either reduces the number of 2-cells or applies the associativity of a monad, we know the number of reductions is linear in the number of natural transformations. Moreover, since checking if a reduction is possible at height i is done in constant time, checking if a reduction is possible at a step is done in linear time, rendering the reduction algorithm quadratic in the number of natural transformations. Since we need to *right* normalize before and after this method, and that this is done in linear time, our theorem is proved. ■

3 Semantic Parsing

In this section we explain our algorithms and heuristics for efficient parsing with as little non-determinism as possible, and reducing time complexity of our parsing strategy.

3.1 Naïve Semantic Parsing on Syntactic Parsing

In this section we suppose that we have a set of syntax trees (or parsing trees) corresponding to a certain sentence. We will now focus on how to derive proofs of typing from a syntax tree. First, note that (?) provides a way to do so by constructing semantic trees labelled by sequence of *combinators*. In our formalism, this amounts to constructing proof trees by mapping combination modes to their equivalent proof trees, inverting if needed the order of the presuppositions to account for the order of the words. This is easily done in linear time in the number of nodes in the parsing tree (which is linear in the input size, more on that in the next section), multiplied by a constant whose size depends on the size of the inference structure. The main idea is that to each node of the tree, both nodes have a type and there is only a finite set of rules that can be applied, provided by the following rules, which are a rewriting of Table 1. In Table 2 we provide *matching-like* rules for different possibilities on the types to combine and the associated possible proof tree(s). Note that there is no condition on what the types *look like*, they can be effectful. If multiple cases are possible, all different possible proof trees should be added to the set of derivations: the set of proof trees for the union of cases is the union of set of proof trees for each case, naturally:

$$PT(\cup C) = \cup PT(C)$$

Remember again that the order of presuppositions for an inference structure does not matter, so always we have the other order of arguments available.

It is important to note that here the proof trees are done “in the wrong direction”, as they are written top-down instead of bottom-up in the sense that we start by the axioms which are the typing judgements of constants in the lexicon (this could actually be said to be a part of the sentence, but non-determinism in the meanings is fine as long as the syntactic category is provided in the parsing tree). This leads to proof trees being a bit weird to the type theorist and a bit weird to the linguist as they are not written as usual trees. Moreover, while a proof tree only provides a type, the tree also provides the sequence of function applications that is needed to compute the actual denotation.

This leads the induced algorithm (for computing the set of denotations) to be exponential in the input in the worst case, when using the recursive scheme that naturally arises from the definition of PT . Indeed, and while this may seem weird, in the case of a function $\mathbf{F}a \rightarrow b$ where \mathbf{F} is applicative and the other combinator is of type $\mathbf{F}a$, one might apply the function directly or apply it under the \mathbf{F} of the argument, leading to two different proof trees:

$$\begin{array}{c}
 \Gamma \vdash \mathbf{F} : \mathcal{C} \Rightarrow \mathcal{C} \quad \frac{\Gamma \vdash \mathbf{F} : \mathcal{C} \Rightarrow \mathcal{C} \quad \Gamma \vdash l = \mathbf{F}l' : \mathbf{F}(a \rightarrow b)}{\Gamma \vdash lr : \mathbf{F}b} \quad \frac{\Gamma \vdash l' : a \rightarrow b \quad \Gamma \vdash r : a}{\Gamma \vdash l'r : b} \text{---pure/return} \\
 \hline
 \Gamma \vdash \mathbf{F}lr : \mathbf{F}b \\
 \hline
 \Gamma \vdash l : \mathbf{F}a \rightarrow b \quad \Gamma \vdash r : \mathbf{F}a \text{---App} \\
 \hline
 \Gamma \vdash lr : b
 \end{array}$$

Those two different proof trees have different semantic interpretations that may be useful, as discussed by ?, especially in their analysis of closure properties, which is modified in the following section.

	S_1	S_2	$PT(S_1, S_2)$
Base Types	$l : \mathbf{a} \rightarrow \mathbf{b}$	$r : \mathbf{a}$	$\frac{\frac{\dots\dots\dots\top}{\Gamma \vdash l : \mathbf{a} \rightarrow \mathbf{b}} \quad \frac{\dots\dots\dots\top}{\Gamma \vdash r : \mathbf{a}}}{\Gamma \vdash lr : \mathbf{b}} \text{App}$
	$l : \mathbf{a} \rightarrow \mathbf{t}$	$r : \mathbf{a} \rightarrow \mathbf{t}$	$\frac{\frac{\dots\dots\dots\top}{\Gamma \vdash l : \mathbf{a} \rightarrow \mathbf{t}} \quad \frac{\dots\dots\dots\top}{\Gamma \vdash r : \mathbf{a} \rightarrow \mathbf{t}}}{\Gamma \vdash l \wedge r : \mathbf{a} \rightarrow \mathbf{t}} \wedge$
Functors	$\Gamma \vdash l : \mathbf{F}\mathbf{a}$	$\Gamma \vdash r : \mathbf{a} \rightarrow \mathbf{b}$	$\frac{\frac{\dots\dots\dots\top}{\Gamma \vdash \mathbf{F} : \mathcal{C} \Rightarrow \mathcal{C}} \quad \frac{\dots\dots\dots\top}{\Gamma \vdash l = \mathbf{F}l' : \mathbf{F}\mathbf{a}} \quad \frac{\frac{\dots\dots\dots\top}{\Gamma \vdash l' : \mathbf{a}} \quad \frac{\dots\dots\dots\top}{\Gamma \vdash r : \mathbf{a} \rightarrow \mathbf{b}}}{\Gamma \vdash rl' : \mathbf{b}}}{\Gamma \vdash lr : \mathbf{F}\mathbf{b}} \text{fmap}$
Applicative	$\Gamma \vdash l : \mathbf{F}(\mathbf{a} \rightarrow \mathbf{b})$	$\Gamma \vdash r : \mathbf{a}$	$\frac{\frac{\dots\dots\dots\top}{\Gamma \vdash \mathbf{F} : \mathcal{C} \Rightarrow \mathcal{C}} \quad \frac{\dots\dots\dots\top}{\Gamma \vdash l = \mathbf{F}l' : \mathbf{F}(\mathbf{a} \rightarrow \mathbf{b})} \quad \frac{\frac{\dots\dots\dots\top}{\Gamma \vdash l' : \mathbf{a} \rightarrow \mathbf{b}} \quad \frac{\dots\dots\dots\top}{\Gamma \vdash r : \mathbf{a}}}{\Gamma \vdash l'r : \mathbf{b}}}{\Gamma \vdash lr : \mathbf{F}\mathbf{b}} \text{pure/return}$

Table 2: List of possible combinations for different presuppositions for inputs, as a definition of a function PT from proof trees to proof trees.

3.1.1 Islands

? provide a formal analysis of islands¹⁷ based on a islands being a different type of branching nodes in the syntactic tree of a sentence, which asks to resolve all **C** effects¹⁸ before that node or being resolved at that node. The main example they propose is the one of existential closure inside conditioning. To reconcile this inside our type system, we propose the following change to their formalism: once the syntactic information of an island existing is added to the tree (or at semantic parsing time, this does not change the time complexity), we *mark* each node inside the island by adding an “void effect” to it, in the same way as we did for our model of plural. This translates into a functor which just maintains the *island marker* on **fmap** and add to the words which create an island node¹⁹ a function which handles the **C** effects, which could be seen as adding a node in the semantics parse tree from the syntactic parse tree. A way to do this would be the following: we pass all functors until finding a **C**, handle it inside the other functors and keep going, on both sides, where PT is defined in Table 2:

$$\text{PT} \left(\frac{\Gamma \vdash x_1 : \mathbf{FCF}'\tau \quad \Gamma \vdash x_2 : \mathbf{FCF}'\tau}{\Gamma \vdash x_1 : \mathbf{FF}'\tau \quad \Gamma \vdash x_2 : \mathbf{FF}'\tau} \right)$$

Note that this preserves the linear size of the parse tree in the number of input words, as we at most double the number of nodes, and note that this would be preserved with additional similar constructs.

This idea amounts to seeing islands, whatever their type, as a form of grammatical/syntactic effect, which is still part of the language but not of the lexicon, a bit like future, modality or plurals, without the semantic component. The idea of seeing everything as effects, while semantically void, allows us to translate into our theory of type-driven semantics outer information from syntax, morphology or phonology that influences the semantics of the sentence. Other examples of this can be found in the modelisation of plural (for morphology, see Sections 1.1.3.3 and 4.2.4) and the emphasis of the words by the **F** effect (for phonology), and show the empirical well-foundedness of this point of view. While we do not aim to provide a theory of everything linguistics through categories²⁰, the idea of expressing everything in our effect-driven type-driven theory of semantics allows us to prepare for any theoretical or empirical observation that has an impact on the semantics of a word/sentence, the allowed combinators and even the addition of rules.

3.2 Rewriting Rules

Here we provide a rewriting system on derivation trees that allows us to improve our time complexity. In the worst case, in big o notation there is no improvement, but there is no loss. The goal is to reduce branching in the definition of *PT*.

First, let’s consider the reductions proposed in Section 2. Those define normal forms under Theorem 2.2 and thus we can use those to reduce the number of trees. Indeed, we usually want to reduce the number of effects by first using the rules *inside* the language, understand, the adjunction co-unit and monadic join. Thus, when we have a presupposition of the form:

$$\frac{}{\Gamma \vdash x : \mathbf{MM}\tau} \quad \text{or} \quad \frac{}{\Gamma \vdash x : \mathbf{RL}\tau}$$

we always simplify it directly, as not always doing it would propagate multiple trees. This in turn means we always verify the derivational rules we set up in the previous section for the join equations of the monad.

¹⁷Syntactic structures which prevent some notion of moving outside of the island.

¹⁸Those represent continuations.

¹⁹Or *post-compose* the proof trees with a rule on handling the **C** effect.

²⁰“A theory is as large as its most well understood examples”.

A TikZ PICTURE GOES HERE.

Figure 2: Usual Typings for some Syntactic Categories

A TikZ PICTURE GOES HERE.

Figure 3: λ -calculus representation of the english language \mathcal{L}

4 Language Translation

In this section we will give a list of words along with a way to express them as either arrows or endo-functors of our typing category. This will also give a set of functors and constructs in our language.

4.1 Syntax

We don't talk about syntactic production rules or any system of the sort: it exists, we take it for granted. The only discussion of translating syntax will be found in Appendix A.2.

4.2 Lambda-Calculus

In this section we use the traditional lambda-calculus denotations.

4.2.1 Types of Syntax

We first need to setup some guidelines for our denotations, by asking ourselves how the different components of sentences interact with each other. For syntactic categories, the types that are generally used are the following, and we will see it matches with our lexicon, and simplifies our functorial definitions²¹. Those are based on \mathbf{T} . Here Υ is the operator which retrieves the type from a certain syntactic category.

4.2.2 Lexicon: Semantic Denotations for Words

Many words will have basically the same “high-level” denotation. For example, the denotation for most common nouns will be of the form: $\Gamma \vdash \lambda x. \mathbf{planet} x : \mathbf{e} \rightarrow \mathbf{t}$. In Table 3 we give a lexicon for a subset of the english language. We describe the constructor for the functors used by our denotations in the table, but all functors will be reminded and properly defined in Table 4 along with their respective \mathbf{fmap} . Note that for terms with two lambdas, we also say that inverting the lambdas also provides a valid denotation. This is important for our formalism as we want to be able to add effects in the order that we want.

4.2.3 Effects of the Language

For the applicatives/monads in Table 4 we do not specify the unit and multiplication functions, as they are quite usual examples. We still provide the \mathbf{fmap} for good measure.

²¹We don't consider effects in the given typings.

A TikZ PICTURE GOES HERE.

Figure 4: Denotations for the functors used

A TikZ PICTURE GOES HERE.

Figure 5: (Partial) Definition for the Π Plural Functor

Let us explain a few of those functors: \mathbf{G} designates reading from a certain environment of type \mathbf{r} while \mathbf{W} encodes the possibility of logging a message of type \mathbf{t} along with the expression. The functor \mathbf{M} describes the possibility for a computation to fail, for example when retrieving a value that does not exist (see **the**). The \mathbf{S} functor represents the space of possibilities that arises from a non-deterministic computation (see **which**).

We can then define an adjunction between \mathbf{G} and \mathbf{W} using our definitions:

$$\varphi : (\alpha \rightarrow \mathbf{G}\beta) \rightarrow \mathbf{W}\alpha \rightarrow \beta \quad \text{and} \quad \psi : (\mathbf{W}\alpha \rightarrow \beta) \rightarrow \alpha \rightarrow \mathbf{G}\beta$$

$$\varphi : \lambda k. \lambda (a, g). k a g \quad \text{and} \quad \psi : \lambda c \lambda a \lambda g. c(a, g)$$

where $\varphi \circ \psi = \text{id}$ and $\psi \circ \varphi = \text{id}$ on the properly defined sets. Now it is easy to see that φ and ψ do define a natural transformation and thus our adjunction $\mathbf{W} \dashv \mathbf{G}$ is well-defined, and that its unit η is $\psi \circ \text{id}$ and its co-unit ε is $\varphi \circ \text{id}$.

As a reminder, this means our language canonically defines a monad $\mathbf{G} \circ \mathbf{W}$. Moreover, the co-unit of the adjunction provides a canonical way for us to deconstruct entirely the comonad $\mathbf{W} \circ \mathbf{G}$, that is we have a natural transformation from $\mathbf{W} \circ \mathbf{G} \Rightarrow \text{Id}$.

4.2.4 Modality, Plurality, Aspect, Tense

We will now explain ways to formalise higher-order concepts as described in Section 1.1.3.3.

First, let us consider in this example the plural concept. Here we do not focus on whether our denotation of the plural is semantically correct, but simply on whether our modelisation makes sense. As such, we give ourselves a way to measure if a certain x is plural or not, which we will denote by $|x| \geq 2^{22}$. It is important to note that if τ is a type, and $\Gamma \vdash x : \tau$ then we should have $|\Pi x| \geq 1 = 2^{23}$. We then need to define the functor Π which models the plural. We do not need to define it on types in any way other than $\Pi(\tau) = \Pi\tau$. We only need to look at the transformation on arrows²⁴. This one depends on the *syntactic*²⁵ type of the arrow, as seen in Table 3. There is actually a presupposition in our definition. Whenever we apply Π to an arrow representing a predicate $p : \mathbf{e} \rightarrow \mathbf{t}$, we still apply p to its argument x even though we say that Πp is the one that applies to a plural entity x . This slight notation abuse results from our point of view on plural/its predicate representation: we assume the predicate p (the common noun, the adjective, the VP...) applies to an object without regarding its cardinality. The two point of views on the singular/plural distinction could be adapted to our formalism: whether we believe that singular is the *natural* state of the objects or that it is to be always specified in the same way as plural does not change anything: In the former, we do not change anything from the proposed functors. In the latter we simply need to create another functor Σ with basically the same rules that represents the singular and ask of our predicates p to be defined on $(\Pi\star) \sqcup (\Sigma\star)$.

See that our functor only acts on the words which return a boolean by adding a plurality condition, and is simply passed down on the other words. Note however there is an issue with the **NP** category: in the case where a **NP** is constructed from a determiner and a common noun (phrase) the plural is not passed down. This comes from the fact that in this case, either the determiner or the common noun (phrase) will be marked, and by functoriality the plural will go up the tree.

²²We chose this representation as an example, it is not important for our formalism that this denotation is actually a good choice for the plural.

²³This might be seen as a typing judgement !

²⁴ \mathbf{fmap} , basically.

²⁵Actually it depends on the word considered, but since the denotations (when considered effect-less) provided in Table 3 are more or less related to the syntactic category of the word, our approximation suffices.

Now clearly our functor verifies the identity and composition laws and works as theorised in Section 1.1.3.3. To understand a bit more why this works as described in our natural transformation rules, consider Π (**which**). The result will be of type $\Pi(\mathbf{S}(e)) = (\Pi \circ \mathbf{S})(e)$. However, when looking at our transformation rule (and our functorial rule) we see that the result will actually be of type $\mathbf{S}(\Pi e)$ which is exactly the expected type when considering the phrase **which** p . The natural transformation θ we set is easily inferable:

$$\theta_A : \begin{array}{ccc} (\Pi \circ \mathbf{S}) A & \xrightarrow{\theta_A} & (\mathbf{S} \circ \Pi) A \\ \Pi(\{x\}) & \longmapsto & \{\Pi(x)\} \end{array}$$

The naturality follows from the definition of **fmap** for the \mathbf{S} functor. We can easily define natural transformations for the other functors in a similar way: $\Pi \circ \mathbf{F} \xRightarrow{\theta} \mathbf{F} \circ \Pi$ defined by $\theta_\tau = \mathbf{fmap}_{\mathbf{F}}(\Pi)$.

Now, in quite a similar way, we can create functors from any sort of judgement that can be seen as a function $\mathbf{e} \rightarrow \mathbf{t}$ in our language²⁶. Indeed, we simply need to replace the $|p| \geq 1$ in most of the definitions by our function and the rest would stay the same. Remember that our use of natural transformations is only there to allow for possible under-markings of the considered effects and propagating the value resulting from the computation of the high-order effect.

4.3 Other Representation of Concepts

In this section we provide explanations on how to translate other semantic descriptions of the lexicon inside our extended type system.

4.3.1 Manifold Representation

$\mathbf{?}$ proposes a way to describe the words of the lexicon (concepts, actually) as manifold.



4.3.2 Vector Representation

$\mathbf{?}$ provides a way to translate vector embeddings to a typed-lambda-calculus system, and as such to a cartesian closed category, meaning we could easily apply the formalism described in Section 4.2.

4.3.3 Probabilistic Monad

$\mathbf{?}$

Acknowledgements

Thanks to Antoine Groudiev for his precious insights on the direction the snakes for () and () should face.

²⁶An easy way to define those would be, similarly to the plural, to define a series of judgement from a property that could be inferred.

A Other Considered Things

A.1 Typing with a Product Category (and a bit of polymorphism)

Another way to start would be to consider product categories: one for the main type system and one for the effects. Let \mathcal{C}_0 be a closed cartesian category representing our main type system. Here we again consider constants and full computations as functions $\perp \rightarrow \tau$ or $\tau \in \text{Obj}(\mathcal{C}_0)$. Now, to type functions and functors, we need to consider a second category: We consider \mathcal{C}_1 the category representing the free monoid on $\mathcal{F}(\mathcal{L})$. Monads and Applicatives will generate relations in that monoid. To ease notation we will denote *functor types* in \mathcal{C}_1 as lists written with head on the left.

Finally, let $\mathcal{C} = \mathcal{C}_0 \times \mathcal{C}_1$ be the product category. This will be our typing category. This means that the real type of objects will be $(\perp \rightarrow \tau, \boxed{})$, which we will still denote by τ . We will denote by $F_n \cdots F_0 \tau$ the type of an object, as if it were a composition of functions²⁷.

In that paradigm, functors simply append to the head of the *functor type* (with the same possible restrictions as before, though I do not see what they would be needed for) while functions will take a polymorphic form: $x : L\tau_1 \mapsto \varphi x : L\tau_2$ and φ 's type can be written as $\star\tau_1 \rightarrow \star\tau_2$.

A.2 Enhanced Grammar

²⁷It is!