On a Category-Theoretic Type and Effect Inference Structure for Semantic Denotation Combinations in Natural Languages: Constructing a Purely Functional Semantic Parser of English using a form of Diagrammatic Calculus integrated in Minimalistic Coproduct-based Merge Theories

Matthieu Boyer

June 26, 2025

Contents

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 Bumford and Charlow (2025).

Introduction

Moggi (1989) 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 Heim and Kratzer (1998).

This is not the first time a categorical representation of a compositional semantics of natural language is proposed, Coecke et al. (2010) 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 CFG[i] of the language.

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

---

[i] Or another model that could generate our language.

Disclaimer

This paper is meant to be read on a computer. Many diagrams, equations and proof trees are two small when printed out to be readable. However reciprocally, many pdf viewers will have trouble rendering the boxes around theorems and some tables, because why not. The author rejects any responsibility in brain or eye damage caused by the reading of this paper, neither in its quality nor in its contents.

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.

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)[ii]. $\bar{\mathcal{C}}$ will be our typing category (in a way).

[ii] Our 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.

We consider for our types the quotient set $\star = \mathrm{Obj}\left(\bar{\mathcal{C}}\right)/\mathcal{F}\left(\mathcal{L}\right)$. Since $\mathcal{F}\left(\mathcal{L}\right)$ does not induce an equivalence relation on $\mathrm{Obj}\left(\bar{\mathcal{C}}\right)$ 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 Melliès and Zeilberger (2015)). We also define $\star_0$ to be the set obtained when considering types which have not yet been *affected*, that is $\mathrm{Obj}(\mathcal{C})$. 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 $\bot \to \tau$ 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 $F\left(A \Rightarrow B\right)$ and $FA \Rightarrow FB$.

We will provide in Figure 15 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*[iii] 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. $\mathcal{F}\left(L\right)\left(\tau \in \star\right)$. 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 typing judgements:

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

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 $\star \to \star$ 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:

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

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

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 $\mathrm{Id}_{\mathcal{C}} \Rightarrow L \circ R$ and $R \circ L \Rightarrow \mathrm{Id}_{\mathcal{C}}$. 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 sequential[iv] computation of the effects added to the

---

[iii]The elements of our language, not the categorical construct.

[iv]In particular, non-necessarily commutative

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 $f : \tau_1 \to \tau_2$ 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 Wadler (1989). This will guarantee for us that given a *Haskell* construction for a polymorphic function, we will get the associated natural transformation.

1.1.3.1 Adjunctions

We will not go in much details about adjunctions, as a full example and generalization process is provided in B.2.3. First, we remind the definition: an adjunction $L \dashv R$ is a pair of functors $L : \mathcal{A} \to \mathcal{B}$ and $R : \mathcal{B} \to \mathcal{A}$, and a pair of natural transformations $\eta : \mathrm{Id}_{\mathcal{A}} \Rightarrow R \circ L$ and $\varepsilon : L \circ R \Rightarrow \mathrm{Id}_{\mathcal{A}}$ such that the two following equations are satisfied: 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 introduce by Maršík and Amblard, 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 Bauer and Pretnar (2014), or Plotkin and Pretnar (2013). As considered by Wu et al. (2014) and van den Berg and Schrijvers (2024), 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 (van den Berg and Schrijvers, 2024) 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, $h \circ \eta = \mathrm{id}$.

What does this mean in our typing category? It means that either our language or our parser for the language [v] should contain natural transformations $F \Rightarrow \mathrm{Id}$ for $F \in \mathcal{F}(\mathcal{L})$. In this goal, we remember that from any polymorphic function in *Haskell* we get a natural transformation (Wadler, 1989) 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. Mathematically[vi], 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 B.2.3) that arises from the phrase *A chair*. This usual example of the differences between the cognitive representation of

---

[v] Depending whether we think handling effects is an intrinsic construct of the semantics of the language or whether it is associated with a speaker.
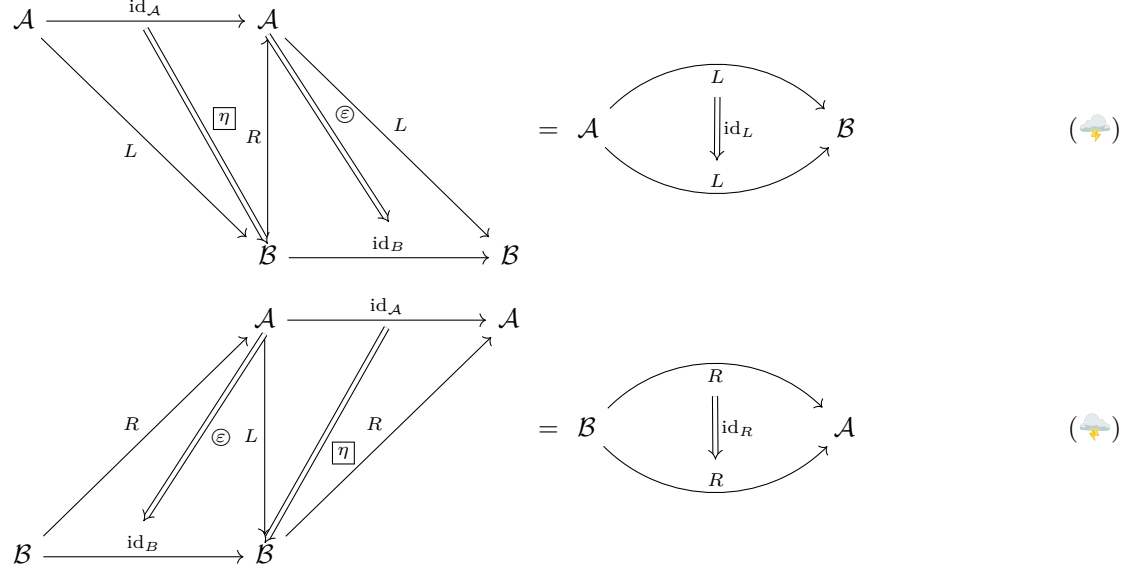
[vi] Computer-wise, actually.

Figure 1: Zig-Zag (⛈) and Zag-Zig (⛈) equations defining adjunctions

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 effects[vii], 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.

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.

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**[viii]. As it can only be applied on adjectives, we can assume its argument is a function (but the definition would hold anyway taking $\tau_1 = \bot$). It is associated with the following function (which is a rewriting of the natural transformation rule):

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

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 recon-

---

[vii]Which we identify to their functorial representations, which we may identify to their free monad in the framework (van den Berg and Schrijvers, 2024).

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

ciliate the following semantic equation even if some of the words are not marked properly:

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

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

$$\mathbf{future}\left(\mathbf{be}\right)\left(\mathbf{arg_1}, \mathbf{arg_2}\right) = \mathbf{future}\left(\mathbf{be}\right)\left(\mathbf{arg_2}\right)\left(\mathbf{future}\left(\mathbf{arg_1}\right)\right) = \mathbf{future}\left(\mathbf{be}\right)\left(\mathbf{future}\left(\mathbf{arg_2}\right)\right)\left(\mathbf{future}\left(\mathbf{arg_1}\right)\right)$$

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.

1.1.3.4   Monad Transformers

In (Bumford and Charlow, 2025), the authors present constructions which they call monad transformers or *higher-order constructors* and which take a monad as input and return a monad as output. One way to type those easily would be to simply create, for each such construct, a monad (the result of the application to any other monad) and a natural transformation which mimics the application and can be seen as the constructor.

1.2   Typing Judgements

To complete this section, Figure 2 gives a simple list of different typing composition judgements through which we also re-derzive 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 infering 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 example: *The man sees the girl using a telescope* in Figure 3.

$$\frac{\Gamma \vdash x : \tau \qquad \Gamma \vdash F : S \subseteq \star \qquad \overbrace{\tau \in S}^{\exists \tau' \in S, \tau \preceq \tau'}}{\Gamma \vdash Fx : F\tau \preceq \tau} \text{Cons}$$

$$\frac{\Gamma \vdash x : \tau \qquad \tau \in \star_0}{\Gamma \vdash Fx : F\tau \notin \star_0} FT_0$$

$$\frac{\Gamma \vdash x : F\tau_1 \qquad \Gamma \vdash \varphi : \tau_1 \to \tau_2}{\Gamma \vdash \varphi x : F\tau_2} \texttt{fmap}$$

$$\frac{\Gamma \vdash x : \tau_1 \qquad \Gamma \vdash \varphi : \tau_1 \to \tau_2}{\Gamma \vdash \varphi x : \tau_2} \text{App}$$

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

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : A\tau} \texttt{pure}$$

$$\frac{\Gamma \vdash x : MM\tau}{\Gamma \vdash x : M\tau} \texttt{>>=}$$

$$\forall F \overset{\theta}{\Longrightarrow} G, \frac{\Gamma \vdash x : F\tau \qquad \Gamma \vdash G : S' \subseteq \star \qquad \tau \in S'}{\Gamma \vdash x : G\tau} \texttt{nat}$$

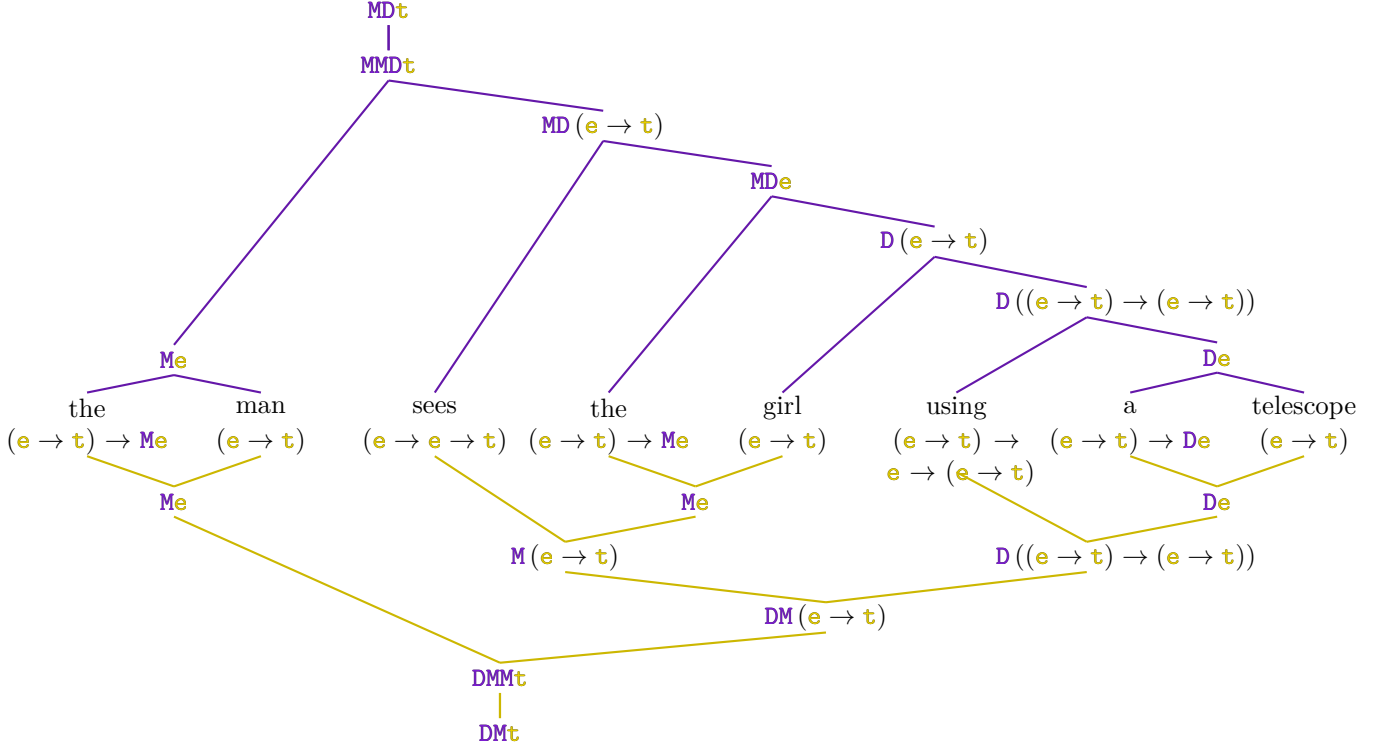Figure 2: Typing and Subtyping Judgements

Figure 3: Parsing trees for the typing of *The man sees the girl using a telescope.*

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 provide methods in Sections 2 and 3 to 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. It is also important to note that we want to be able to map effects in any possible order and as such, we did not provide all the possible typings for this sentence, see Section 3 for more details.

1.3 Decidability

The observant reader might have notice (without too much trouble), that our typing system is not decidable, because of the `nat/pure/return` rules which may allow for infinite derivations. The good news is, this is only an issue when trying to type things that cannot be typed. Indeed, because of the considerations on handling, and the fact that semantically void units will in the end get deleted without any modifications on the end meaning, we can just remove the rules that allow for unbounded derivations, especially the usage of units of applicatives and natural transformations that allow to switch back and forth between two or more functors. This leads to derivations of sentences to be of bounded height, linear in the length of the sentence.

2 Handling Non-Determinism

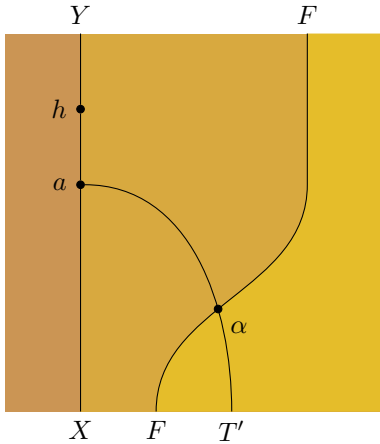The typing judgements proposed in Section 1.2 lead to ambiguity. In this section we propose ways to get our derivations to a certain normal form, by deriving an equivalence relation on our derivation and parsing trees, based on string diagrams.

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
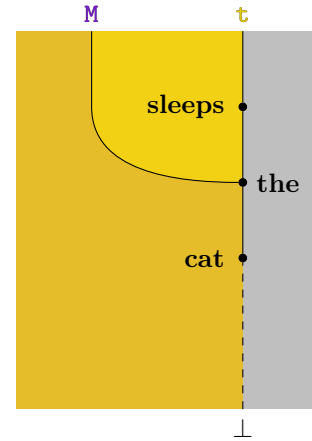
categories and functors. In this category, the endofunctors are natural transformations. This means that we represent categories as regions of the plane, functors as lines separating regions and natural transformations as the intersection points between two lines. We will change the color of the regions 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:

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 $\mathbb{1} \to \mathcal{C}$ 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 $\tau_1 \to \tau_2$. 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.

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 (Joyal and Street, 1991) 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 $=$) and yielding a quotient set of *normal forms* for our computations.

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 currifiable, 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 obvious[ix]. The natural composition rules on diagrams being the gluing together of diagrams, we do

---

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

need to add something to our toolbox.
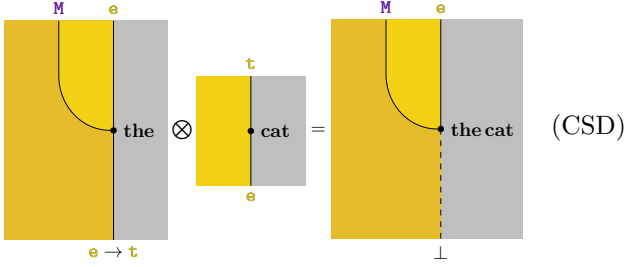
To justify our proposition to only consider fully reduced expressions note that in this formalism we don't consider the expressions for our functors and natural transformations[x] but simply the sequence in which they are applied. In particular, this means the following diagrams commute for any F, G 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 touch[xi]. 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).

---

[x]Here, 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…

[xi]It looks nice and makes you forget this is applied abstract nonsense.
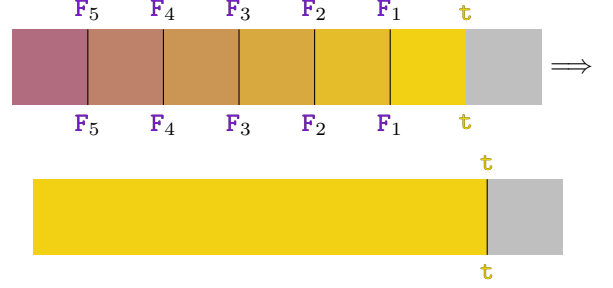
It is important to acknowledge that a previous modelisation of parsers by string diagrams has already been proposed (Coecke et al. (2010), Wang-Maścianica (2023)). We will develop on that in Section 3.2.2. We keep the work on vertically composing string diagrams as a way to see how the effects pile up, 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 `Ma → b`. Indeed, (Bumford and Charlow, 2025) suggests that those could be of type `St → t` and that we might want to apply that function to something of type `St` by first using the unit of the monad `S` then invoking `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 reconciliate 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 *Currying* rule:

$$\text{(Currying)}$$

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 diagram[xii] to another effectful diagram? Take the equation presented in Figure 4.

Note that the effects are applied in the order of the application, meaning that if we apply a function $\varphi : \tau_1 \to \tau_2 \to \tau_3$ 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 Joyal and Street (1991) 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 (Selinger, 2010), Theorem 1.2 (Joyal and Street, 1991)** 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.

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.

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

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:

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 $L \dashv R$:

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

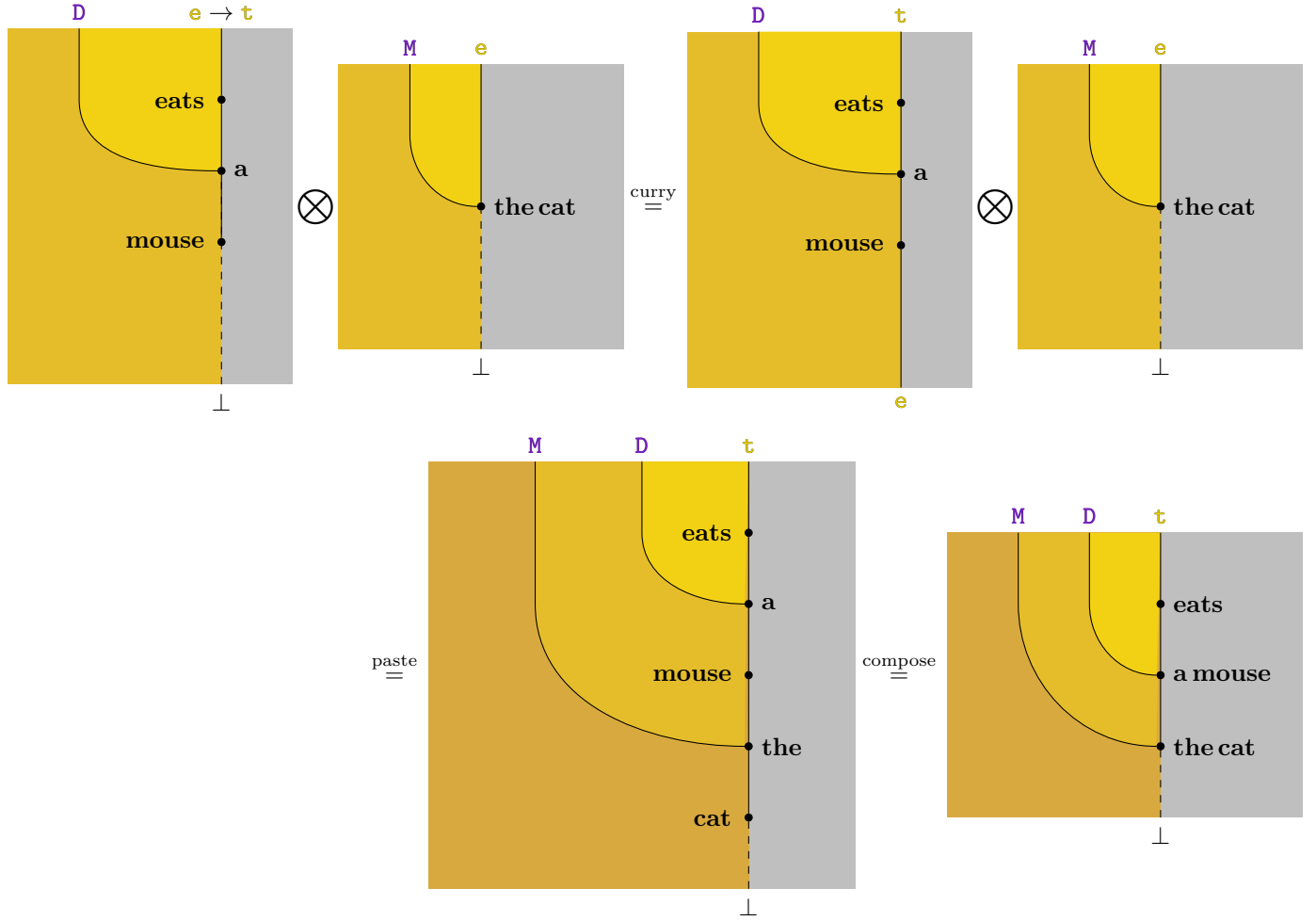What the reduction means is that we have a commutative mapping:

Figure 4: Example of easy composition of kleisli arrows in the string diagram format, with no derivation concerns.

The *(co-)Monadic* Equations explain the very familiar sentence that a monad is just[xiii] 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.
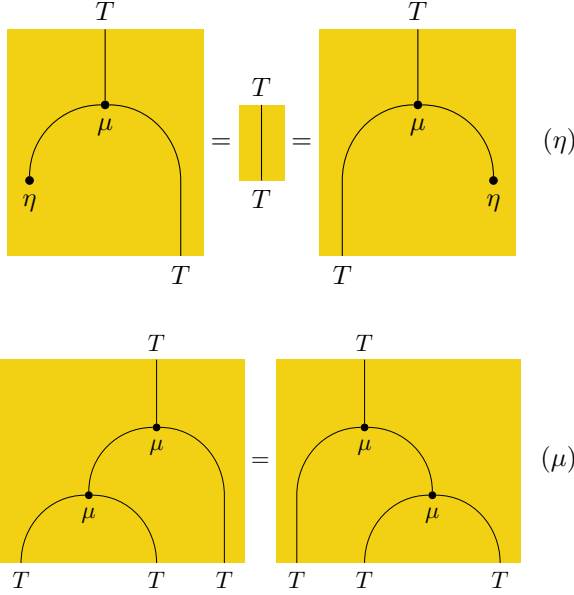
This set of equations, when added to our composition rules from Section 2.2.1 explain all the different reductions that can be made to limit nondeterminism in our parsing strategies. Indeed, considering the equivalence relation $\mathcal{R}$ freely generated from $\left\{ (\text{📅}), (\text{🐛}), (\text{🐛}), (\mu), (\eta) \right\}$ 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:

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

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.

Delpeuch and Vicary (2022) 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 Delpeuch and Vicary (2022) 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 elements[xiv]: $(D.N, D.S, D.L)$ 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 $nat = (nat.\text{h}, nat.\text{in}, nat.\text{out})$. From this, we propose a naive algorithm to check if a string diagram is valid or not: Note that here the algorithm does not require that all effects are handled for validity.

[xiii]Which is a monad

[xiv]We could write it as a tuple of 5 elements by replacing $D.\text{L}$ by three functions that lower level

---

Algorithm 1 Validity Check

function ISVALID(D)
    $S \leftarrow D.\text{S}$
    for $i < D.\text{N}$ **do**
        $nat \leftarrow D.\text{L}(i)$
        $b, e \leftarrow nat.\text{h}, nat.\text{h} + |nat.\text{in}|$
        if $S[b : e] \neq nat.\text{in}$ **then return** False
        $S[b : e] \leftarrow nat.\text{out}$▷ To understand as replacing
in place.
        return True

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 Delpeuch and Vicary (2022). 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 $\mathcal{O}\left(n \times \sup_i |D.\text{L}(i).\text{in}| + |D.\text{L}(i).\text{out}|\right)$, 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 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:

- $D.\text{L}(i).\text{h} = D.\text{L}(i+1).\text{h} - 1$

- $D.\text{L}(i) = \eta_{L,R}$

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

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 $\mu\left(\mu\left(TT\right), T\right) \rightarrow \mu\left(T\mu\left(TT\right)\right)$ 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 Delpeuch and Vicary (2022). 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. Since right reductions do not modify the equational reductions, and thus right reducing an equational normal form yields

an equational normal form, combining the two systems is done without issue, 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. ∎

2.3.3 Completeness, Soundness, Decidability

Let us do a short sanity check on the well-foundedness of our reduction system. Clearly from its definition this equivalence system is sound, in that all the reductions we allow are based on actual equivalence rules, and thus each reduction step preserves the equivalence relationship. From the previous Theorem 2.3, clearly, our system is decidable, since it's in P. However, there is no reason that our system would be complete: At this point, we possibly might not have the widest sound reduction system: there might be more reduction rules that can apply and that would augment the equivalence classes under the reduction relationship. Moreover, there is the possibility that the widest sound reduction system is not computable. This means that our current systems are decidable, sound but not complete.

3 Efficient Semantic Parsing

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

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 (Bumford and Charlow, 2025) 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. Computing one tree 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 Figure 2. In Figure 5 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:

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

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

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

Figure 5: List of possible combinations for different presuppositions for inputs, as a definition of a function *PT* from proof trees to proof trees.

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 $\mathbf{F}\mathbf{a} \to \mathbf{b}$ where $\mathbf{F}$ is applicative and the other combinator is of type $\mathbf{F}\mathbf{a}$, one might apply the function directly or apply it under the $\mathbf{F}$ of the argument, leading to two different proof trees:

$$\dfrac{\overline{\Gamma \vdash \mathbf{F} : \mathcal{C} \Rightarrow \mathcal{C}} \quad \dfrac{\overline{\Gamma \vdash \mathbf{F} : \mathcal{C} \Rightarrow \mathcal{C}}^{\top} \quad \overline{\Gamma \vdash l = \mathbf{F}l' : \mathbf{F}(\mathbf{a} \to \mathbf{b})}^{\top} \quad \dfrac{\overline{\Gamma \vdash l' : \mathbf{a} \to \mathbf{b}}^{\top} \quad \overline{\Gamma \vdash r : \mathbf{a}}^{\top}}{\Gamma \vdash l'r : \mathbf{b}}}{\Gamma \vdash lr : \mathbf{F}\mathbf{b}} \texttt{pure/return}}{\Gamma \vdash \mathbf{F}lr : \mathbf{F}\mathbf{b}}$$

$$\dfrac{\overline{\Gamma \vdash l : \mathbf{F}\mathbf{a} \to \mathbf{b}}^{\top} \quad \overline{\Gamma \vdash r : \mathbf{F}\mathbf{a}}^{\top}}{\Gamma \vdash lr : \mathbf{b}} \text{App}$$

Those two different proof trees have different semantic interpretations that may be useful, as discussed by Bumford and Charlow (2025), especially in their analysis of closure properties, which is modified in the following section.

3.1.1 Islands

Bumford and Charlow (2025) provide a formal analysis of islands[xv] based on a islands being a different type of branching nodes in the syntactic tree of a sentence, which asks to resolve all $\mathbf{C}$ effects[xvi] 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 node[xvii] a function which handles the $\mathbf{C}$ 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 $\mathbf{C}$, handle it inside the other functors and keep going, on both sides, where PT is defined in Figure 5:

$$\text{PT}\left(\dfrac{\Gamma \vdash x_1 : \texttt{FCF'}\tau}{\Gamma \vdash x_1 : \texttt{FF'}\tau} \,\middle|\, \dfrac{\Gamma \vdash x_2 : \texttt{FCF'}\tau}{\Gamma \vdash x_2 : \texttt{FF'}\tau}\right)$$

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

---

[xv]Syntactic structures which prevent some notion of moving outside of the island.

[xvi]Those represent continuations.

[xvii]Or *post-compose* the proof trees with a rule on handling the $\mathbf{C}$ effect.

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 B.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 categories[xviii], 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 Syntactic-Semantic Parsing

3.2.1 The Improved Method

If using a naïve strategy on the trees yields an exponential algorithm, the best way to improve the algorithm, is to directly leave the naïve strategy. To do so, we could simply extend the grammar system used to do the syntactic part of the parsing. In this section, we will take the example of a CFG since it suffices to create our typing combinators, For example, we can increase the alphabet to be the free monoid on the product Σ of the usual base alphabet and $\bar{\mathcal{C}}$ our typing category. Since the usual syntactic structure and our combination modes can both be expressed by CFGs (or almost, cf Figure 6), our product system can be expressed by (almost) a CFG. Here, we change our notations from the proof trees of Figure 5 to have a more explicit grammar of combination modes provided below is a rewriting of the proof trees provided earlier, and highlights the combination modes in a simpler way, based on (Bumford and Charlow, 2025) as it simplifies the rewriting of our typing judgements in a CFG[xix]. The grammars provided in Figures 14 and 6 are actually used from right to left, as we actually do combinations over the types and syntactic categories of the words and try to reduce the sentence, not derive it from nothing. Now, all the improvements discussed in Section 3.3 can still be applied here, just a bit differently, as this amounts to reducing ambiguity in the grammar.

This grammar works in five major sections:

$$\tau \quad ::= \quad \tau \Rightarrow \tau$$
$$\quad | \quad \langle \tau, \tau \rangle$$
$$\quad | \quad \bar{\tau}$$
$$\quad | \quad F\tau$$

$$\bar{\tau} \quad ::= \quad \mathtt{t} \mid \mathtt{e} \mid \cdots$$

$$F \quad ::= \quad \mathcal{F}(\mathcal{L})^*$$

$$\mathrm{Tree} \quad ::= \quad \tau, \tau$$

$$SM \quad ::= \quad \mathrm{ML} \mid \mathrm{MR}$$
$$\quad | \quad \mathrm{UL} \mid \mathrm{UR}$$
$$\quad | \quad \mathrm{A}$$
$$\quad | \quad \mathrm{J}$$
$$\quad | \quad \mathrm{C}$$
$$\quad | \quad \mathrm{ER} \mid \mathrm{EL}$$
$$\quad | \quad \mathrm{DN}$$

$$\mathrm{M} \quad ::= \quad SM, M$$
$$\quad | \quad \mathrm{BComb}$$

$$\mathrm{BComb} \quad ::= \quad > \mid < \mid \wedge \mid \vee$$

$$>, \beta \quad ::= \quad (\alpha \to \beta), \alpha$$
$$<, \beta \quad ::= \quad \alpha, (\alpha \to \beta)$$
$$\wedge, \alpha \to \mathtt{t} \quad ::= \quad (\alpha \to \mathtt{t}), (\alpha \to \mathtt{t})$$
$$\vee, \alpha \to \mathtt{t} \quad ::= \quad (\alpha \to \mathtt{t}), (\alpha \to \mathtt{t})$$

$$\mathrm{J}_{\mathbf{F}} \; \mathbf{F}\tau \quad ::= \quad \mathbf{F}\mathbf{F}\tau$$
$$\mathrm{DN}_{\mathbf{C}} \; \tau \quad ::= \quad \mathbf{C}_\tau \tau$$

$$\mathrm{ML}_{\mathbf{F}}(\alpha, \beta) \quad ::= \quad \mathbf{F}\alpha, \beta$$
$$\mathrm{MR}_{\mathbf{F}}(\alpha, \beta) \quad ::= \quad \alpha, \mathbf{F}\beta$$
$$\mathrm{A}_{\mathbf{F}}(\alpha, \beta) \quad ::= \quad \mathbf{F}\alpha, \mathbf{F}\beta$$
$$\mathrm{UR}_{\mathbf{F}}(\alpha \to \alpha', \beta) \quad ::= \quad \mathbf{F}\alpha \to \alpha', \beta$$
$$\mathrm{UL}_{\mathbf{F}}(\alpha, \beta \to \beta') \quad ::= \quad \alpha, \mathbf{F}\beta \to \beta'$$
$$\mathrm{C}_{\mathbf{LR}}(\mathbf{L}\alpha, \mathbf{R}\beta) \quad ::= \quad (\alpha, \beta)$$
$$\mathrm{ER}_{\mathbf{R}}(\mathbf{R}(\alpha \to \alpha'), \beta) \quad ::= \quad \alpha \to \mathbf{R}\alpha', \beta$$
$$\mathrm{EL}_{\mathbf{R}}(\alpha, \mathbf{R}(\beta \to \beta')) \quad ::= \quad \alpha, \beta \to \mathbf{R}\beta'$$

Figure 6: Possible Type Combinations in the form of a near CFG

[xviii]*"A theory is as large as its most well understood examples"*.

[xix]That, in a sense, was already implicitly provided in Figure 2.

1. We reintroduce the grammar defining the type and effect system.

2. We introduce a structure for the semantic parse trees and their labels, the combination modes from Bumford and Charlow (2025).

3. We introduce rules for basic type combinations.

4. We introduce rules for higher-order unary type combinators.

5. We introduce rules for higher-order binary type combinators.

The idea of the *grammatical* reduction is that from the flat sentence, we create a full parse tree as a sequence of types τ. We then reduce it using the binary effect combinators, before choosing the appropriate binary type combinator. It is at this point in the reduction we actually do the composition of functions. We close up the reduction[xx] by possibly using unary effect combinators.

We do not prove here that these typing rules cannot be written in a simpler syntactic structure[xxi], but it is easy to see why a regular expression would not work, and since we need trees, to express our full system, the best we can do would be to disambiguate the grammar. A thing to note is that this grammar is not complete but explains how types can be combined in a compact form. For example, the rules of the form $\mathrm{ML}_{\mathbf{F}} M, \tau' \leftarrow M, \tau$ are rules that provide ways to combine effects from the two inputs in the order we want to: we can combine `RSe` and `CW(e → t)` into `RCWSt` with the mode $\mathrm{MLMRMRML} <$ (see (Bumford and Charlow, 2025) Example 5.14).

Each of these combinators can be, up to order, associated with a inference rule, and, as such, with a higher-order denotation, which explains the actual effect[xxii] of the combinator, and are described in Figure 7. The main reason we need to get denotations associated to combinators, is to properly define equivalence and reduce the number of rules in the grammar. Explanation on how that is done will come in Section 3.3. The important thing on those derivation is that they're a direct translation of the rules defining the notions of functors, ap-

$$>= \lambda\varphi.\lambda x.\varphi x$$

$$<= \lambda x.\lambda\varphi.\varphi x$$

$$\mathrm{ML}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathbf{F}}\lambda a.M(a,y))x$$

$$\mathrm{MR}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathbf{F}}\lambda b.M(x,b))y$$

$$\mathrm{A}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathbf{F}}\lambda a.\lambda b.M(a,b))(x)\mathtt{<*>}y$$

$$\mathrm{UL}_{\mathbf{F}} = \lambda M.\lambda x.\lambda\varphi.M(x, \lambda b.\varphi(\eta_{\mathbf{F}}b))$$

$$\mathrm{UR}_{\mathbf{F}} = \lambda M.\lambda\varphi.\lambda y.M(\lambda a.\varphi(\eta_{\mathbf{F}}a), y)$$

$$\mathrm{J}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.\mu_{\mathbf{F}}M(x,y)$$

$$\mathrm{C}_{\mathbf{LR}} = \lambda M.\lambda x.\lambda y.\varepsilon_{\mathbf{LR}}(\mathtt{fmap}_{\mathbf{L}}(\lambda l.\mathtt{fmap}_{\mathbf{R}}(\lambda r.M(l,r))(y))(x))$$

$$\mathrm{EL}_{\mathbf{R}} = \lambda M.\lambda\varphi.\lambda y.M(\Upsilon_{\mathbf{R}}\varphi, y)$$

$$\mathrm{ER}_{\mathbf{R}} = \lambda M.\lambda x.\lambda\varphi.M(x, \Upsilon_{\mathbf{R}}\varphi)$$

$$\mathrm{DN}_{\Downarrow} = \lambda M.\lambda x.\lambda y. \Downarrow M(x,y)$$

Figure 7: Denotations describing the effect of the combinators used in the grammar describing our combination modes presented in Figure 6
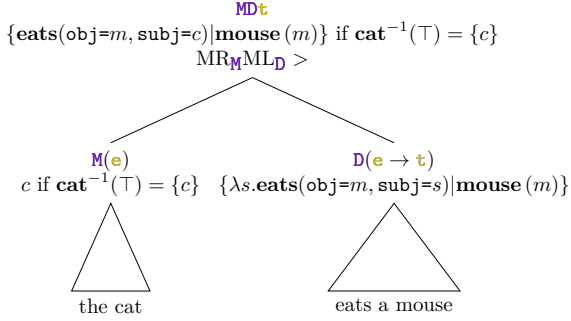
plicatives, monads and thus are not specific to any denotation system, even though we will use lambda-calculus styled denotations to describe them. The same structure for combinators apply when describing the combinators than when describing their rules of existence. This makes us able to compute the actual denotations associated to a sentence using our formalism, as presented in figure 8. Note that the order of combination modes is not actually the same as the one that would come from the grammar. The reason why will become more apparent when string diagrams for parsing are introduced in the next section, but simply, this comes from the fact that while we think of ML and MR as reducing the number of effects on each side (and this is the correct way to think about those), this is not actually how its denotation works, but there is no real issue with that.

Moreover, it is important to note that while we talk about the structure in Figure 6 as a grammar, it is more of a structure that computes if it is possible or not to combine two types and how. In particular, our grammar is not finite, since there are infinitely many types that can be playing the part of α and β, and infinitely many functors that can play the roles for $\mathbf{F}$ and so on, but that does not pose a problem, as recognizing the general form of a type can be done in constant time for the forms that we want to check, given a proper memory representation.
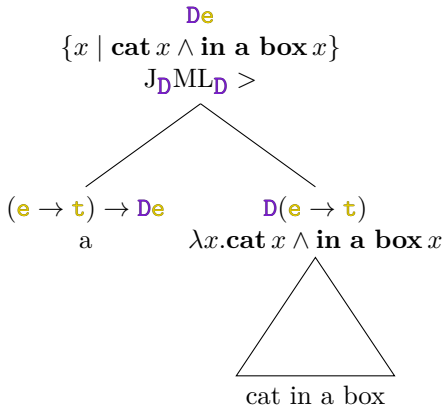

[xx]For the combination of two words, which is then repeated along the syntactic structure.

[xxi]It is important to note that for a typing system, we only have syntactic-like rules that allow, or not, to combine types.
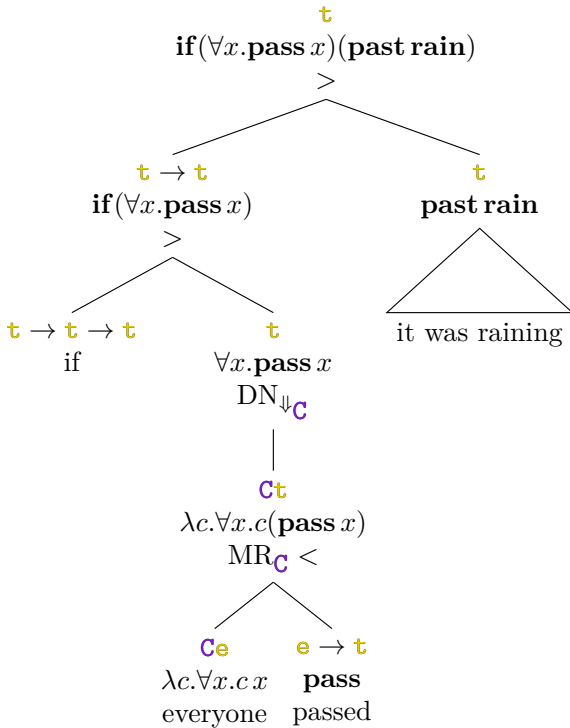
[xxii]Pun intended

MDt
$\{\mathbf{eats}(\mathtt{obj}{=}m, \mathtt{subj}{=}c)|\mathbf{mouse}(m)\}$ if $\mathbf{cat}^{-1}(\top) = \{c\}$
$\mathrm{MR_M ML_D} >$

- **M(e)**
  c if $\mathbf{cat}^{-1}(\top) = \{c\}$
  — the cat
- **D(e → t)**
  $\{\lambda s.\mathbf{eats}(\mathtt{obj}{=}m, \mathtt{subj}{=}s)|\mathbf{mouse}(m)\}$
  — eats a mouse

(a) Labelled tree representing the equivalent parsing diagram to 10

De
$\{x \mid \mathbf{cat}\,x \wedge \mathbf{in\ a\ box}\,x\}$
$\mathrm{J_D ML_D} >$

- $(\mathtt{e} \to \mathtt{t}) \to$ **De**
  a
- **D(e → t)**
  $\lambda x.\mathbf{cat}\,x \wedge \mathbf{in\ a\ box}\,x$
  — cat in a box

(b) Labelled tree representing the equivalent parsing diagram to 11

t
$\mathbf{if}(\forall x.\mathbf{pass}\,x)(\mathbf{past\ rain})$
$>$

- **t → t**
  $\mathbf{if}(\forall x.\mathbf{pass}\,x)$
  $>$
  - **t → t → t**
    if
  - **t**
    $\forall x.\mathbf{pass}\,x$
    $\mathrm{DN_{\Downarrow C}}$
    |
    Ct
    $\lambda c.\forall x.c(\mathbf{pass}\,x)$
    $\mathrm{MR_C} <$
    - **Ce**
      $\lambda c.\forall x.c\,x$
      everyone
    - **e → t**
      pass
      passed
- **t**
  $\mathbf{past\ rain}$
  — it was raining

(c) Labelled tree representing the equivalent parsing diagram to 13

Figure 8: Examples of Labeled Parse Trees for a few sentences.

Furthermore, it can easily be rendered into something that looks like a Chomsky Normal Form for a CFG and allows us to integrate this in the CYK algorithm, and even get a correct time complexity. Since our grammar is not actually finite, the modified version of CYK that parses only the types (not the full system) will have a complexity in the size of $\mathcal{F}(\mathcal{L})$ that is linear, albeit with a not so small constant factor, which comes from the fact that our grammar's size is linear in $|\mathcal{F}(\mathcal{L})|$.

> **Theorem 3.1** Semantic parsing of a sentence is polynomial in the length of the sentence and the size of the type system and syntax system.

Proof. Suppose we are given a syntactic generating structure G_s along with our type combination grammar G_τ. The system G that can be constructed from the (tensor) product of G_s and G_τ has size $|G_s| \times |G_\tau|$. Indeed, we think of its rules as a rule for the syntactic categories and a rule for the type combinations. Its terminals and non terminals are also in the cartesian products of the adequate sets in the original grammars. What this in turn means, is that if we can syntactically parse a sentence in polynomial time in the size of the input and in $|G_s|$, then we can syntactico-semantically parse it in polynomial time in the size of the input, $|G_s|$ and $|G_\tau|$. ∎

While we have gone with the hypothesis that we have a CFG for our language, any type of polynomial-time structure could work[xxiii], as long as it is more expressive than a CFG. We will do the following analysis using a CFG since it allows to cover enough of the language for our example and for simplicity of describing the process of adding the typing CFG, even though some think that English is not a context free language (Higginbotham, 1984). Since the CYK algorithm provides us with an algorithm for CFG based parsing cubic in the input and linear in the grammar size, we end up with an algorithm for semantically parsing a sentence that is cubic in the length of the sentence and linear in the size of the grammar modeling the language and the type system.

> **Theorem 3.2** Retrieving a pure denotation for a sentence can be done in polynomial time in the length of

---

[xxiii]Simply, the algorithm will need to be adapted to at least process the CFG for the typing rules.

the sentence, given a polynomial time syntactic pars-
ing algorithm and polynomial time combinators.

Proof. We have proved in Theorem 3.1 that we can re-
trieve a semantic parse tree from a sentence in polyno-
mial time in the input. Since we also have shown that
the length of a semantic parse tree is quadratic in the
length of the sentence it represents, being linear in the
length of a syntactic parse tree linear in the length of
the sentence. We have already seen that given a deno-
tation, handling all effects and reducing effect handling
to normal forms can be done in polynomial time. The
superposition of these steps yields a polynomial-time al-
gorithm in the length of the input sentence.   ■

The *polynomial time combinators* assumption is not
a complex assumption, this is for example true for our
denotations in Section B, with function application being
linear in the number of uses of variables in the function
term, which in turn is linear in the number of terms
used to construct the function term and thus of words,
and `fmap`being in constant time[xxiv] for the same reason.
Of course, function application could be said to be in
constant time too, but we consider here the duration of
the β-reduction:

$$(\lambda x.M)\, N \xrightarrow{\beta} M\,[x/N]$$

3.2.2   Diagrammatical Parsing

When considering Coecke et al. (2010) way of using
string diagrams for syntactic parsing/reductions, we can
see them as (yet) another way of writing our parsing
rules. In our typed category, we can make see our combi-
nators as natural transformations (2-cells): then we can
see the different sets of combinators as different arity nat-
ural transformations. $>$, $ML_{\mathbf{F}}$ and $J_{\mathbf{F}}$ are represented in
Figure 9, up to the coloring of the regions, because that
is purely for an artistic rendition[xxv].

Now, a way to see this would be to think of this as an
orthogonal set of diagrams to the ones of Section 2: we
could use the syntactic version of the diagrams to model
our parsing, according to the rules in Figure 6, and then
combine the diagrams as in Equation CSD, as shown in
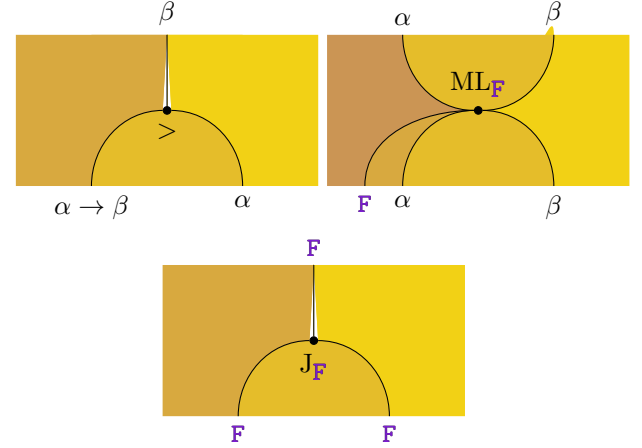Figure 10, which highlights why we can consider the di-

Figure 9: String Diagramatic Representation of
Combinator Modes $>$, ML and J

agrams orthogonals. The obvious reaction[xxvi] is: "*Why
did we start by saying we could just paste?*". This is a
perfectly valid point, and pasting diagrams is simply a
notation abuse, that is justified by the fact there is a
possible reduction. In this figure we exactly see the se-
quence of reductions play out on the types of the words,
and thus we also see what exact *quasi-braiding* would
be needed to construct the effect diagram. Here we talk
about *quasi-braiding* because, in a sense, we use 2-cells to
do braiding-like[xxvii] operations on the strings, and don't
actually allow for braiding inside the diagrammatic com-
putation. To better understand what happens in those
parsing diagrams, Figure 8 provides the translations in
labelled trees of the parsing diagrams of Figures 10, 11
and 13.

Categorically, what this means is that we start from a
meaning category $\mathcal{C}$, our typing category, and take it as
our grammatical category. This is a form of extension
on the monoidal version by Coecke et al. (2010), as it
is seemingly a typed version, where we change the Pre-
group category for the typing category, possibly taken
with a product for representation of the English gram-
mar representation, to accomodate for syntactic typing
on top of semantic typing. This is, again, just another
rewriting of our typing rules.

More formally, we have a first axis of string diagrams
in the category $\mathcal{C}$ - our string diagrams for effect han-
dling, as in Section 2 - and a second *orthogonal* axis

[xxiv]Depending on the functor but still bounded by a constant.
[xxv]Colours make this less boring, trust me

[xxvi]That I'm sure everyone have had seeing this.
[xxvii]Permutations on the order of the strings

of string diagrams on a larger category, with endofunctors[xxviii] graded by the types in our typing category $\bar{\mathcal{C}}$ and with natural transformations mapping the combinators defined in Figures 6 and 7. The category in which we consider the second-axis string diagrams does not have a meaning in our compositional semantics theory, and to be more precise, we should talk about 1-cells and 2-cells instead of functors and natural transformations, to keep in the idea that this is really just a diagrammatic way of computing and presenting the operations that are put to work during semantic parsing.

What the lines leading from combinators to functors[xxix] mean categorically, is void in either category. Those lines are not actually part of the first axis of the string diagram, nor are they part of the second axis of the string diagram. They are used to map out the link between the two: they express the quasi-braiding step proposed above, and present graphically why the order of the strings in the resulting diagram is as it is, and what the pasting and quasi-braiding orders should be. A good approach to these diagrams might be the following: they are an extension of the parse trees presented in Bumford and Charlow (2025), applied to the formalism of Coecke et al. (2010) and extended to be integrated with the handling of effects, as described in Section 2, forming a full diagrammatic calculus system for semantic parsing.

For the combinators J, DN and C, which are applied to reduce the number of effects inside a denotation, it might seem less obvious how to include them. While this may seem true, it's actually only true for the *connecting* strings. Indeed, applying them to the actual *parsing* part of the diagram is done in the exact same way as in the CFG, we just apply them when needed, and they will appear in the resulting denotation as an end for a string, a form of forced handling, in a sense, as shown in the result of Figure 11. For the connecting strings, it's simply a matter of adding a new *phantom* string that will send the associated 2-cell in the effect handling diagram to the connected strings. In particular it is interesting to note that the resulting diagram representing the sentence can, in a way, be found in the connection strings that arise from the combinators.

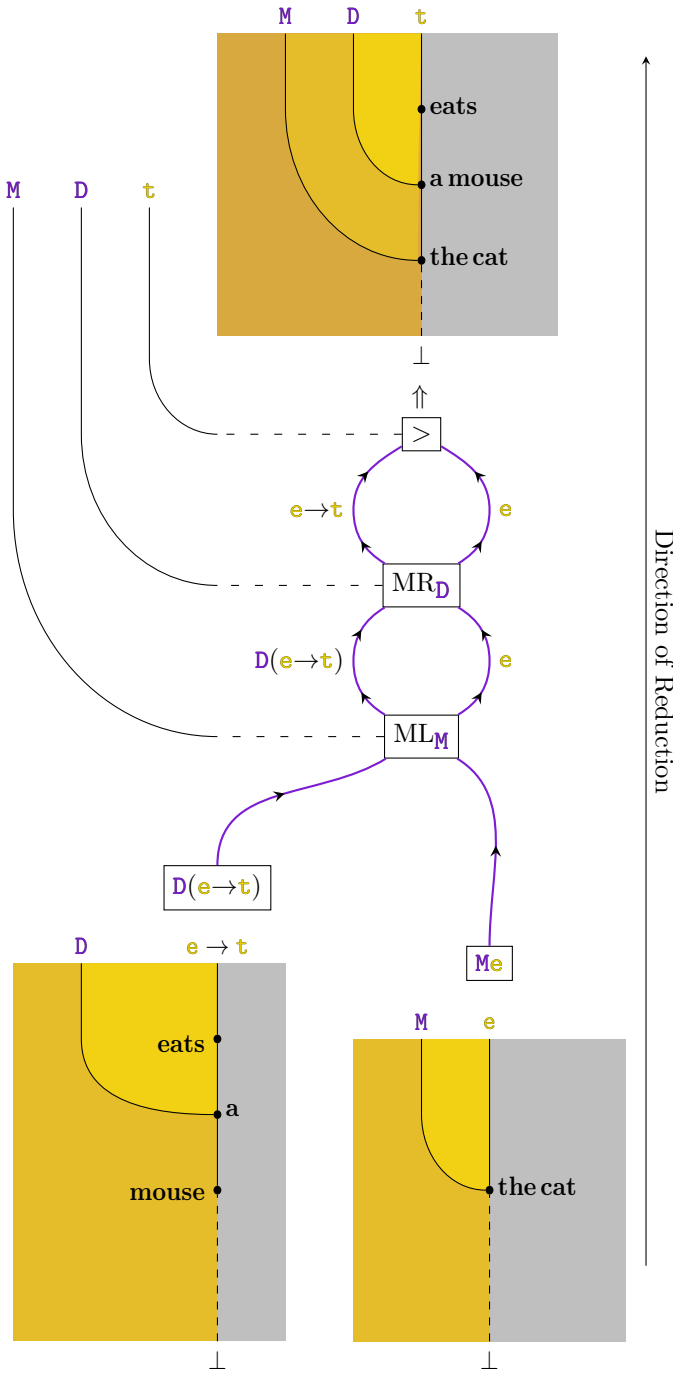The main reason why this point of view of diagram-

Figure 10: Representation of a parsing (sub-)diagram for the sentence *the cat eats a mouse*, including the connection between the effect handling diagrams from Section 2 and the syntactio-semantic parsing diagrams formed combining our approach to parsing and (Coecke et al., 2010) approach to diagrams. See Figure 8b for translation in a parse tree.

[xxviii]To ensure proper typing of the diagrams.

[xxix]Remember that types are objects in $\mathcal{C}$ and thus functors from $\mathbb{1} \Rightarrow \mathcal{C}$
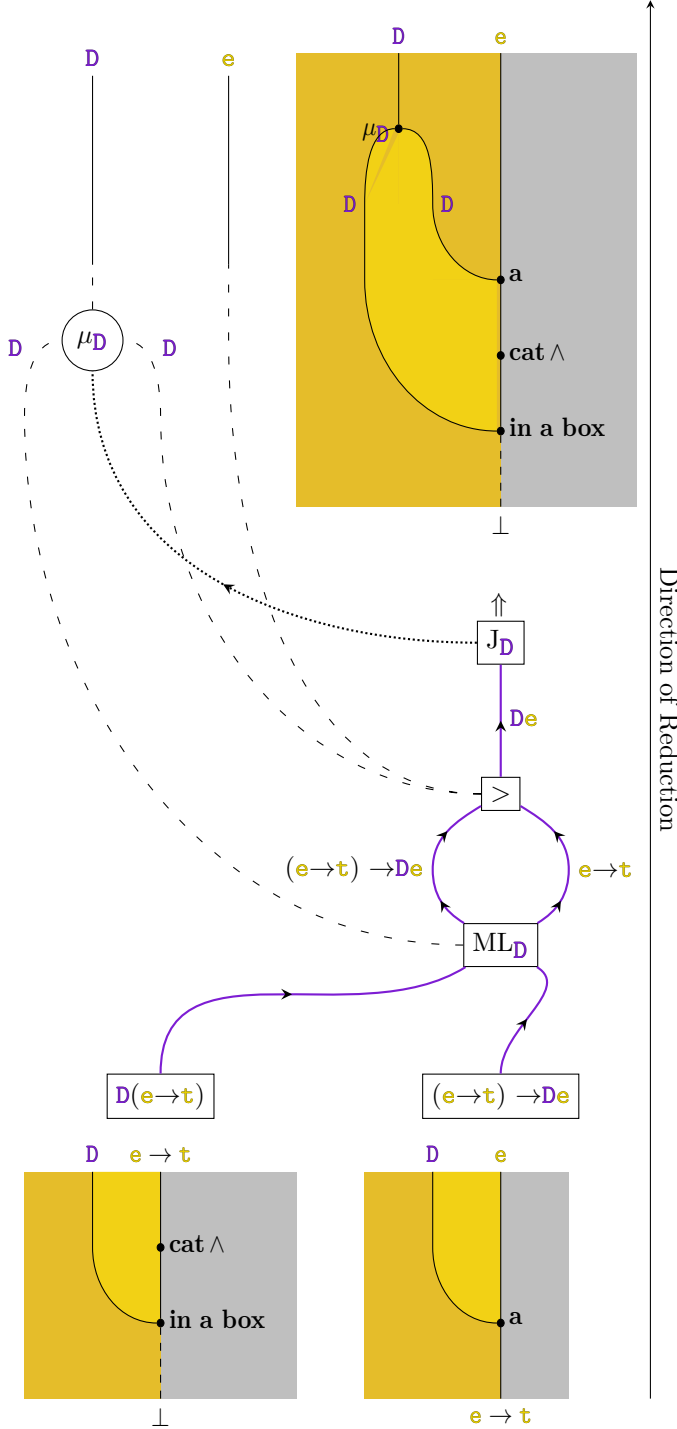
Figure 11: Example of a parsing (sub-)diagram for the phrase *a cat in a box*, presenting the integration of unary combinators inside the connector line. See Figure 8b for translation in a parse tree.

Figure 12: Example of a *Jacquard* knitwork. Photography and work courtesy of the author's mother.

matic parsing is useful will be clear when looking at the rewriting rules and the normal forms they induce, because, as seen before, string diagrams make it easy to compute normal forms, when provided with a confluent reduction system.

The other reason being the tangible interpretation of how things work underlying the idea of a string diagram: Suppose you're knitting a rainbow scarf. You have multiple threads (the different words) of the different colours (their types and effects) you're using to knit the scarf. When you decide to change the color you take the different threads you have been using, and mix them up: You can create a new colour[xxx] thread from two (that's the base combinators) create a thicker one from two of the same colour (the applicative mode and the monadic join), put aside a thread until a later step (that's the `fmap`), add a new thread to the pattern (that's the unit), or cut a thread you will not be using anymore (that's the co-units and closure operators). Changing a thread by cutting it and making a knot at another point is basically what the eject combinators do. This more tangible representation can be seen in a larger diagram in Figure 13. The sections in the rectangle represent what happen when considering our combination step as implementing patterns inside a knitwork, as seen in Figure 12. The different patterns provide, in order, a visual representation of the different ways one can combine two strings, i.e., two types and thus two denotations. The sections outside of the rectangle are the strings of yarn not currently being used to make a pattern.

[xxx]I know this is not how wool works, but if you prefer you can imagine a pointillist-like way of drawing using multiple coloured lines that superimpose on each other, or a marching band's multiple instruments playing either in harmony or in disharmony and changing that during a score.
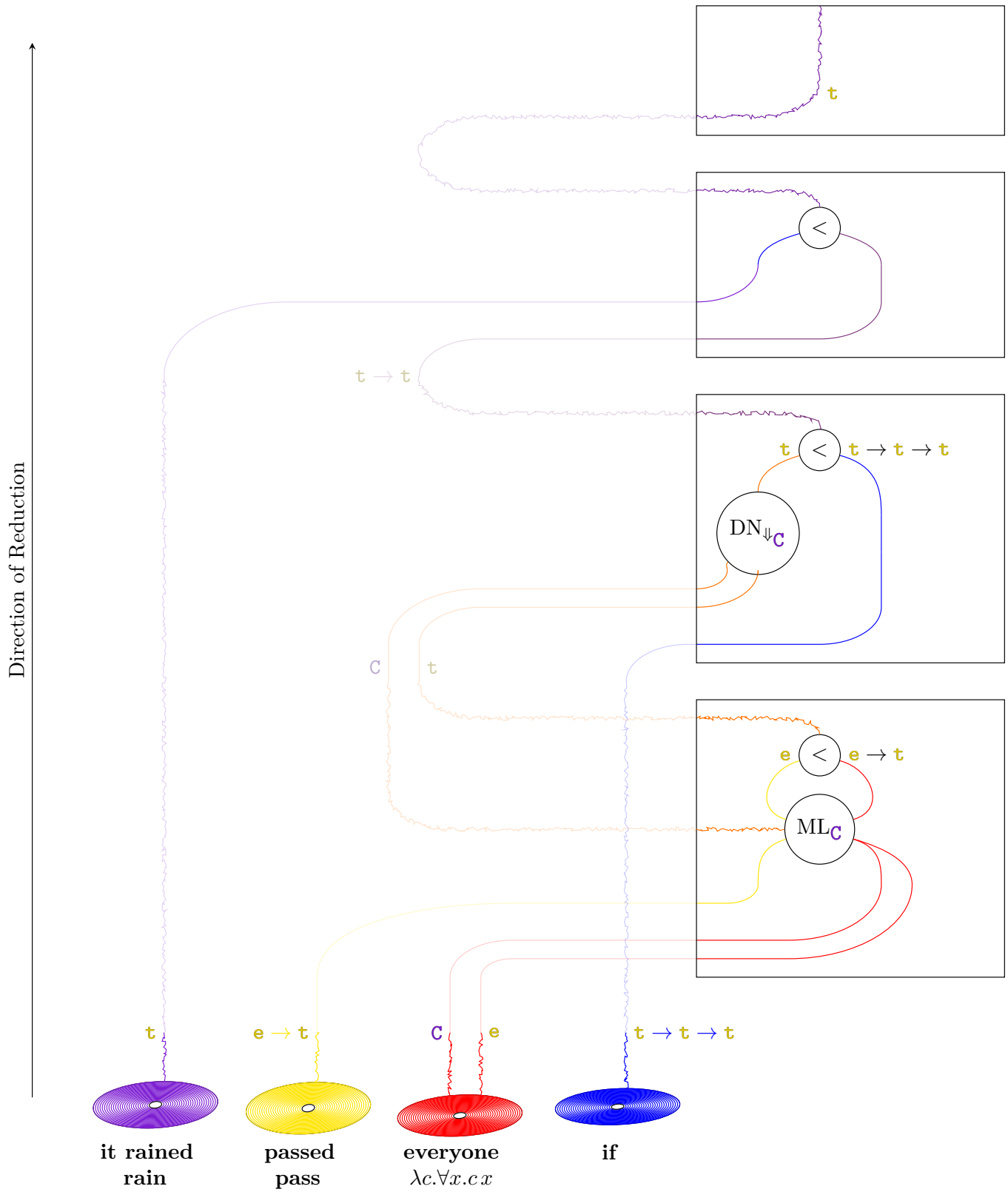
Figure 13: 3D-like Representation of the Diagrammatic Parsing of a Sentence. See 8c for the translation in a Parse Tree

3.2.3   The Coproduct Categorical Approach to Syntax

In this section based on the work by Marcolli, Matilde et Chomsky, Noam et Berwick, Robert C., we explore the integration of our notion inside the structure of syntactic merge, and why the two formalisms are compatible. The idea behind that structure is to see the union of trees as a product and the merging of trees as based on a coproduct in a well-defined Hopf algebra. Now of course, with our insights on syntactico-semantic parsing, we can either think of our parsing structure labeled by multiple modes, as in (Bumford and Charlow, 2025), or think of it as string diagrams, as presented above. In both cases, we make a more or less implicit use of the merge operation.

Labeled trees can be seen as computed from a sequence of labeled merge. As such, it is easy to see how one can adapt the construction of a purely syntactic merge onto a semantic merge, using a similar approach. There is just the need for a coloured operad to create a series of labeled merges, like proposed in (Melliès and Zeilberger, 2025) for example. Since there is a one-to-one mapping from our string diagrams to parse trees, mapping explained when comparing the tables for denotations of combinators in Figure 7 and Figure 12 in (Bumford and Charlow, 2025), or by looking at the parse trees equivalent to string diagrams in Figures 10, 11 and 13, it is easy to see that indeed, there is a notion of merge inside our diagrams that can in a way be expressed through a Hopf algebra, by transporting the diagrams to and from their equivalent parsing diagram. More generally, what this means is that a merge, in our string diagrams, is the addition of a combinator to one, two[xxxi] without conditions. This is useful as a definition, because it allows integration of our system in a broader framework, including for example the notion of morphosyntactic trees.

However, this is not fully satisfying: like for syntax trees, applying a random sequence of merges to a set of input strings will not always yield a properly typed denotation. Since type soundness is the main feature of our system, it seems weird to have a definition of merge which cannot take that into account.

[xxxi]Or more, see C.2

3.3   Rewriting Rules

Here we provide a rewriting system on derivation trees that allows us to improve our time complexity by reducing the size of the disambigued grammar. In the worst case, in big o notation there is no improvement in the size of the sentence, but there is no loss. The goal is to reduce branching in the definition of PT, as this will easily translate into reductions in the grammar.

When considering the grammar version of the semantic system, the reductions written below cannot be done when using only one step at a time. To get around this, a simple way might be to expand the size of the grammar to consider reductions two at a time, by adding an intermediate non-terminal. Then, while the size of the grammar becomes quadratic in $|\mathcal{F}(\mathcal{L})|$, because this also reduces the number of reductions needed to get the derivations, this compensates the increase, and actually reduces the time complexity when the reducing the number of rules after equivalence. Obviously the same reasoning applies when considering reductions of length 3, 4 and more. We really just need to consider the same set of reductions that the one proposed below.

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:

$$\frac{\overline{}}{\Gamma \vdash x : \texttt{MM}\tau}\top \qquad \text{or} \qquad \frac{\overline{}}{\Gamma \vdash x : RL\tau}\top$$

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.

Secondly, while there is no way to reduce the branching proposed in the previous section since they end in completely different reductions, there is another case in which two different reductions arise:

$$\frac{\overline{}}{\Gamma \vdash x : \texttt{F}\tau_1}\top \qquad \text{and} \qquad \frac{\overline{}}{\Gamma \vdash y : \texttt{G}\tau_2}\top$$

Indeed, in that case we could either get a result with effects $\texttt{FG}$ or with effects $\texttt{GF}$. In general those effects are not

the same, but if they happen to be, which trivially will be the case when one of the effects is external, the plural or islands functors for example. When the two functors commute, the order of application does not matter and thus we choose to get the outer effect the one of the left side of the combinator.

Thirdly, there are modes that clearly encompass other ones[xxxii]. One should not use mode UR when using MR or DNMR and the same goes for the left side, because the two derivations yield simpler results. Same things can be said for certain other derivations containing the lowering and co-unit combinators.
We use DN when we have not used any of the following, in all derivations:

- $m_{\mathbf{F}}, \mathrm{DN}, m_{\mathbf{F}}$ where $m \in \{\mathrm{MR}, \mathrm{ML}\}$

- $\mathrm{ML}_{\mathbf{F}}, \mathrm{DN}, \mathrm{MR}_{\mathbf{F}}$

- $\mathrm{A}_{\mathbf{F}}, \mathrm{DN}, \mathrm{MR}_{\mathbf{F}}$

- $\mathrm{ML}_{\mathbf{F}}, \mathrm{DN}, \mathrm{A}_{\mathbf{F}}$

- C

We use J if we have not used any of the following, for $j \in \{\varepsilon, \mathrm{J}_{\mathbf{F}}\}$

- $\{m_{\mathbf{F}}, j, m_{\mathbf{F}}\}$ where $m \in \{\mathrm{MR}, \mathrm{ML}\}$

- $\mathrm{ML}_{\mathbf{F}}, j, \mathrm{MR}_{\mathbf{f}}$

- $\mathrm{A}_{\mathbf{F}}, j, \mathrm{MR}_{\mathbf{F}},$

- $\mathrm{ML}_{\mathbf{F}}, j, \mathrm{A}_{\mathbf{F}}$

- k, C for $k \in \{\varepsilon, \mathrm{A}_{\mathbf{F}}\}$

- If $\mathbf{F}$ is commutative as a monad:

  - $\mathrm{MR}_{\mathbf{F}}, \mathrm{A}_{\mathbf{F}}$

  - $\mathrm{A}_{\mathbf{F}}, \mathrm{ML}_{\mathbf{F}}$

  - $\mathrm{MR}_{\mathbf{F}}, j, \mathrm{ML}_{\mathbf{F}}$

  - $\mathrm{A}_{\mathbf{F}}, j, \mathrm{A}_{\mathbf{F}}$

> **Theorem 3.3** The rules proposed above yield equivalent results.

Proof. For the first point, the equivalence has already been proved under Theorem 2.2. For the second point, it is obvious since based on an equality. For the third point, it's a bit more complicated. The rules about not using combinators UL and UR come from the notion of handling and granting termination and decidability to our system. The rules about adding J and DN after moving two of the same effect from the same side (i.e. MLML or MRMR) are normalization of a the elevator equations 🗓. Indeed, in the denotation, the only reason to keep two of the same effects and not join them[xxxiii] is

to at some point have something get in between the two. Joining and cloture should then be done at earliest point in parsing where it can be done, and that is equivalent to later points because of the elevator equations, or Theorem 2.1. The last set of rules follows from the following: we should not use JMLMR instead of A, as those are equivalent because of the equation defining them. The same thing goes for the other two, as we should use the units of monads over applicative rules and fmap. ∎

The observant reader might have noticed that this is simply a scheme to apply typing rules to a syntactic derivation, but that this will not be enough to actually gain all reductions possible in polynomial time. This is actually not feasible (because of the intrinsic ambiguity of the English language, as proved for example by the sentence *The man sees the girl using a telescope*). Even then, we are far from reducing to a minimum the number of different paths possible to get a same final denotation. This can only happen once a confluent reduction scheme is provided for the denotations. When this is done, we can combine the reduction schemes for effects along with the one for denotation and the one for combinations in one large reduction scheme. Indeed, trivially, the tensor product of confluent reduction schemes forms a confluent reduction scheme, whose maximal reduction length is the sum of the maximal reduction lengths[xxxiv].

When using our diagrammatic approach to parsing, which, again, is just a rewriting in a more graphical fashion of our typing rules and our CFG-like rules, we can write all the reductions described above to our paradigm: it simply amounts to constructing a set of normal forms for the string diagrams. This leads to the same algorithms developed in Section 2 being usable here: we just have a new improved version of Theorem 2.2 which adds the normal forms specified in this section to the newly added *orthogonal* axis of diagrammatic computations. What we're actually doing is computing two different normal forms along the tensor product of our reduction schemes[xxxv], but again, that only amounts to computing a larger normal form. Moreover, considering the possible normal forms of syntactic reductions simply adds another way to reduce our diagrams to normal

---

[xxxii]Here we use the grammar notation for ease of explanation
[xxxiii]Provided they can be joined.

[xxxiv]Actually it is at most that, but that does not really matter.
[xxxv]Just like we already have (or should have) a tensor scheme for the denotations and combination modes.

forms. Since all of these forms can be attained in polynomial time, it is clear that finding a normal-form diagram, which is exactly a normal-form denotation for the sentence, is doable in polynomial time[xxxvi]

Once again, there is no evidence that our system is complete, if not the contrary, so the arguments developed in Section 2.3.3 are still valid. A way to "complete" it, although it would still probably be incomplete would be to write an automatized prover in Lean, but this is out of the scope of this project, as it would not do many improvements.

Acknowledgements

[xxxvi]Of course, this is only true when the denotations can be normalized in polynomial time.

A Bibliography

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4.

Dylan Bumford and Simon Charlow. 2025. Effect-driven interpretation: Functors for natural language composition.

Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. 2010. Mathematical Foundations for a Compositional Distributional Model of Meaning.

Antonin Delpeuch and Jamie Vicary. 2022. Normalization for planar string diagrams and a quadratic equivalence algorithm.

Michael Goodale. 2022. Manifolds as conceptual representations in formal semantics.

Julian Grove and Jean-Philippe Bernardy. 2023. Probabilistic compositional semantics, purely.

Irene Heim and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Number 13 in Blackwell Textbooks in Linguistics. Blackwell, Malden, MA.

James Higginbotham. 1984. English Is Not a Context-Free Language. *Linguistic Inquiry*, 15(2):225–234.

André Joyal and Ross Street. 1991. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112.

Marcolli, Matilde et Chomsky, Noam et Berwick, Robert C. Mathematical Structure of Syntactic Merge.

Jiří Maršík and Maxime Amblard. Algebraic Effects and Handlers in Natural Language Interpretation.

Paul-André Melliès and Noam Zeilberger. 2015. Functors are Type Refinement Systems. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 3–16, Mumbai India. ACM.

Paul-André Melliès and Noam Zeilberger. 2025. The categorical contours of the Chomsky-Sch\"utzenberger representation theorem. *Logical Methods in Computer Science*, Volume 21, Issue 2:13654.

E. Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, USA. IEEE Comput. Soc. Press.

B Partee. Lecture 2. Lambda abstraction, NP semantics, and a Fragment of English. *Formal Semantics*.

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4.

Mehrnoosh Sadrzadeh and Reinhard Muskens. 2018. Static and Dynamic Vector Semantics for Lambda Calculus Models of Natural Language.

Peter Selinger. 2010. A survey of graphical languages for monoidal categories. volume 813, pages 289–355.

Birthe van den Berg and Tom Schrijvers. 2024. A framework for higher-order effects & handlers. *Science of Computer Programming*, 234:103086.

Philip Wadler. 1989. Theorems for free! In *Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture*, FPCA '89, pages 347–359, New York, NY, USA. Association for Computing Machinery.

V. Wang-Maścianica. 2023. *String Diagrams for Text*. http://purl.org/dc/dcmitype/Text, University of Oxford.

Nicolas Wu, Tom Schrijvers, and Ralf Hinze. 2014. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA. Association for Computing Machinery.

B Presenting a Language

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.

B.1 Syntax

In Figure 14 we provide a toy Context-Free Grammar to support the claims made in Section 3.2. The discussion

```
CP    ::=  DP, VP          NP    ::=  AdjP, NP
      |    Cmp, CP                |    NP, AdjP
      |    CP, CBar
                            AdjP  ::=  TAdj, DP
CBar  ::=  Cor, CP                |    Deg, AdjP

Dbar  ::=  Cor, DP          VP    ::=  TV, DP
                                  |    AV, CP
DP    ::=  DP, Dbar               |    VP, AdvP
      |    Dmp, DP
      |    Det, NP          TV    ::=  DV, DP
      |    Gen, TN
                            AdvP  ::=  TAdv, DP
Gen   ::=  DP, GenD
```

Figure 14: Partial CFG for the English Language

of whether this system accurately models the syntax of the english language is far beyond the scope of this paper, but could replace fully this section. Our syntactic categories will be written in a more linguistically appreciated style than the one proposed in Figure 15. Our set of categories is the following:

```
CP, Cmp, CBar, DBar, Cor, DP, Det, Gen, GenD,
Dmp, NP, TN, VP, TV, DV, AV, AdjP, TAdj, Deg,
                 AdvP, TAdv
```

B.2 Lambda-Calculus

In this section we use the traditional lambda-calculus denotations, with two types for truth values and entities freely generating our typing category.

B.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[xxxvii], the types that are generally used are the following, and we will see it matches with our lexicon, and simplifies our functorial definitions[xxxviii]. Those are based on Partee. Here Υ is the operator which retrieves the type from a certain syntactic category.

[xxxvii]Those are a subset of the ones proposed in Section B.1.
[xxxviii]We don't consider effects in the given typings.

S $\texttt{t}$

CN(P) $\texttt{e} \to \texttt{t}$

ProperN $\texttt{e}$

NP Either:

  $\texttt{e}$ *John, a cat*

  $(\texttt{e} \to \texttt{t}) \to \texttt{t}$ *the cat, every man*

DET $\Upsilon\,(\textbf{CN}) \to \Upsilon\,(\textbf{NP})$

ADJ(P) Either:

  $(\texttt{e} \to \texttt{t})$ *carnivorous, happy*

  $(\texttt{e} \to \texttt{t}) \to (\texttt{e} \to \texttt{t})$ *skillful*

REL $\texttt{e} \to \texttt{t}$

VP, IV(P) $\texttt{e} \to \texttt{t}$

TV(P) $\Upsilon\,(\textbf{NP}) \to \Upsilon\,(\textbf{VP})$

is $(\texttt{e} \to \texttt{t}) \to \texttt{e} \to \texttt{t}$

Figure 15: Usual Typings for some Syntactic Categories

B.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: $\Gamma \vdash \lambda x.\textbf{planet}\,x : \texttt{e} \to \texttt{t}$. In Figure 16 we give a lexicon for a subset of the english language, without the syntactic categories associated with each word, since some are not actually words but modifiers on words[xxxix], and since most are obvious but only contain one meaning of a word. We describe the constructor for the functors used by our denotations in the table, but all functors will be reminded and properly defined in Figure 17 along with their respective `fmap`. For denotations that might not be clear at first glance, $\cdot_\texttt{F}$ is the phonologic accentuation or emphasis of a word and $\cdot, \textbf{a} \cdot$ is the apposition of a **NP** to a **DP** or proper noun. 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.

[xxxix]Emphasis, for example.

Expression	Type	λ-Term
planet	$\mathtt{e} \to \mathtt{t}$	$\lambda x.\mathbf{planet}\, x$
	Generalizes to **common nouns**	
carnivorous	$(\mathtt{e} \to \mathtt{t})$	$\lambda x.\mathbf{carnivorous}\, x$
	Generalizes to **predicative adjectives**	
skillful	$(\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t})$	$\lambda p.\lambda x.px \wedge \mathbf{skillful}\, x$
	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	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$	$\lambda p.\lambda x.px$
she	$\mathtt{r} \to \mathtt{e}$	$\lambda g.g_0$
it	$\mathtt{Ge}$	$\lambda g.g_0$
which	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Se}$	$\lambda p.\{x \mid px\}$
the	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Me}$	$\lambda p.x$ if $p^{-1}(\top) = \{x\}$ else $\#$
a	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{De}$	$\lambda p.\lambda s.\{\langle x, x + s\rangle \mid px\}$
no	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Ce}$	$\lambda p.\lambda c.\neg \exists x.px \wedge c\,x$
every	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Ce}$	$\lambda p.\lambda c.\forall x, px \Rightarrow cx$
$\cdot, \mathbf{a}\cdot$	$\mathtt{e} \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{We}$	$\lambda x.\lambda p.\langle x, px\rangle$
as for	$\mathtt{e} \to \mathtt{Te}$	$\lambda x.\lambda s.\langle x, x + s\rangle$
$\cdot_{\mathbf{F}}$	$\mathtt{e} \to \mathtt{Fe}$	$\lambda v.\langle v, \{x \mid x \in D_e\}\rangle$

Figure 16: λ-calculus representation of the english language $\mathcal{L}$

B.2.3   Effects of the Language

For the applicatives/monads in Figure 17 we do not specify the unit and multiplication functions, as they are quite usual examples. We still provide the `fmap` for good measure.

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

We can then define an adjunction between `G` and `W` using our definitions:

$$\varphi : \begin{array}{c} (\alpha \to \mathtt{G}\beta) \to \mathtt{W}\alpha \to \beta \\ \lambda k.\lambda\,(a,g)\,.kag \end{array} \quad \text{and } \psi : \begin{array}{c} (\mathtt{W}\alpha \to \beta) \to \alpha \to \mathtt{G}\beta \\ \lambda c\lambda a\lambda g.c\,(a,g) \end{array}$$

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

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

B.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 our not, which we will denote by $|x| \geq 2^{\mathrm{xl}}$. It is important to note that if τ is a type, and $\Gamma \vdash x : \tau$ then we should have $|\Pi x| \geq 1 = \top^{\mathrm{xli}}$. 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 arrows[xlii]. This one depends on the *syntactic*[xliii] type of the arrow, as seen in Figure 16. There is actually a presupposition in our definition. Whenever we apply Π to an arrow representing a predicate $p : \mathtt{e} \to \mathtt{t}$, 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 adjectve, 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.

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 $\Pi\,(\mathbf{which})$. The result will be of type $\Pi\,(\mathtt{S}\,(e)) = (\Pi \circ \mathtt{S})\,(e)$. However, when looking at our transformation rule (and our functorial rule) we see that the result will actually be of type $\mathtt{S}\,(\Pi e)$ which is exactly the expected type when considering the phrase **which** p. The natural transformation θ we set is easily inferable:

$$\theta_A : \begin{array}{c} (\Pi \circ \mathtt{S})\,A \xrightarrow{\ \theta_A\ } (\mathtt{S} \circ \Pi)\,A \\ \Pi\,(\{x\}) \longmapsto \{\Pi(x)\} \end{array}$$

---

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

xli This might be seen as a typing judgement !

xlii `fmap`, basically.

xliii Actually it depends on the word considered, but since the denotations (when considered effect-less) provided in Figure 16 are more or less related to the syntactic category of the word, our approximation suffices.

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
$\mathtt{C}\,(\tau) = (\tau \to \mathtt{t}) \to \mathtt{t}$	$\mathtt{C}\varphi\,(x) = \lambda c.x\,(\lambda a.c\,(\varphi a))$	Monad
$\mathtt{T}\,(\tau) = \mathtt{s} \to (\tau \times \mathtt{s})$	$\mathtt{D}\varphi\,(\lambda s.\{\langle x, x \mathbin{+\mkern-8mu+} s\rangle \mid px\}) = \lambda s.\,\langle \varphi x, \varphi x \mathbin{+\mkern-8mu+} s\rangle$	Monad
$\mathtt{F}\,(\tau) = \tau \times \mathtt{S}\tau$	$\mathtt{F}\varphi\,(\langle v, \{x \mid x \in D_e\}\rangle) = \langle \varphi\,(v), \{x \mid x \in D_e\}\rangle$	Monad
$\mathtt{D}\,(\tau) = \mathtt{s} \to \mathtt{S}\,(\tau \times \mathtt{s})$	$\mathtt{D}\varphi\,(\lambda s.\{\langle x, x \mathbin{+\mkern-8mu+} s\rangle \mid px\}) = \lambda s.\{\langle \varphi x, \varphi x \mathbin{+\mkern-8mu+} s\rangle \mid px\}$	Monad
$\mathtt{M}\,(\tau) = \tau + \bot$	$\mathtt{M}\varphi\,(x) = \begin{cases} \varphi\,(x) & \text{if } \Gamma \vdash x : \tau \\ \# & \text{if } \Gamma \vdash x : \# \end{cases}$	Monad

Figure 17: Denotations for the functors used

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	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to ((\mathtt{e} \to \mathtt{t}) \to \mathtt{t})$	$\Pi(p) = \lambda \nu.p\,(\Pi\nu)$
	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to \mathtt{e}$	$\Pi(p) = \lambda \nu.p\,(\Pi\nu)$

Figure 18: (Partial) Definition for the Π Plural Functor

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

Now, in quite a similar way, we can create functors from any sort of judgement that can be seen as a function `e` $\to$ `t` in our language[xliv]. 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.

B.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.

B.3.1   Manifold Representation

Goodale (2022) proposes a way to describe the words of the lexicon (concepts, actually) as manifold.

B.3.2   Vector Representation

Sadrzadeh and Muskens (2018) 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 B.2.

B.3.3   Probabilistic Monad

Grove and Bernardy (2023)

C   Other Considered Things

C.1   Typing with a products 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 $\bot \to \tau$ or $\tau \in \mathrm{Obj}(\mathcal{C}_0)$. 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 $(\bot \to \tau, [])$, 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 functions[xlv].

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 $\star\tau_1 \to \star\tau_2$.

C.2   Multiple Ways

TODO: : add explanations on how to modify the thing to take more arities, or whatever the way to define a language is.

[xliv]An easy way to define those would be, similarly to the plural, to define a series of judgement from a property that could be infered.

[xlv]It is!