Formalizing a Functional Programming Language for Natural Language Semantics

Internship Report

Matthieu Boyer

Under the supervision of Simon Charlow

September 9, 2025

Abstract

In this report, we will present a Haskell-inspired type and effect system to model natural language semantics and provide a diagrammatic calculus to help with effect handling and parsing in this formalism. This allows us to implement complex language behaviours such as anaphoras and non-determinism inside a type-and-effect-dependent parser.

Acknowledgements

I want to thank my mother for the knitting and putting up with me peeling potato to try knitting with toothpicks; Antoine Groudiev for his precious insights on the direction in which the snakes for (🐍) and (🐍) should face; Paul-André Melliès for his insights; Bob Frank for the help with the minimalistic merge syntactic theories; and of course, Simon Charlow for his guidance during my time at Yale, and his help around the linguistics questions that definitely arose.

Introduction

What is *a chair*? How do I know that *Jupiter, a planet*, is *a planet*? To answer those questions, [2] provides a Haskell-based view on the notion of typing in natural language semantics. Their main idea is to include a layer of effects which allows for improvements in the expressiveness of the denotations used. This allows us to model complex concepts such as anaphoras, or non-determinism in an easy way, independent of the actual way the words are represented. Indeed, when considering the usual denotations of words as typed lambda-terms, this allows us to solve the issue of meaning getting lost through impossible typing, while still being able to compose meanings properly. When two expressions have the same syntactic distribution, they must also have the same type, which forces quantificational noun phrases to have the same type as proper nouns: the entity type `e`. However, there is no singular entity that is the referent of *every planet*, and so, the type system gets in the way of meaning, instead of being a tool at its service.

Our formalism is inscribed in the contemporary natural language semantic theories based on three main elements: a *lexicon*, a *syntactic description* of the language, and a theory of *composition*. More specifically, we explain how to extend the domain of the lexicon and the theory of composition to account for the phenomena described above. We will not be discussing most of the linguistic foundations for the usage of the formalism, nor its usefulness. We refer to [2] to get an overview of the linguistic considerations at the base of the theory.

In this report, we will provide a formal definition of an enhanced type and effect system for natural language semantics, implementing categorical tools. This will increase the complexity (both in terms of algorithmic operations and in comprehension of the model) of the parsing algorithms, but through the use of string diagrams to model the effect of composition on potentially effectful denotations (or more generally computations), we will provide efficient algorithms for computing the set of meanings of a sentence from the meaning of its components. A mathematical presentation of the english language based on [2] and expansion proposals on this work

can be found in the appendices.

1 Related Work

This is not the first time a categorical representation of compositional semantics of natural language is proposed, [3] already suggested an approach based on monoidal categories using an external model of meaning. Our approach gives additional latitude for the definition of denotations in the lexicon, and a visual explanation of the difference between multiple possible parsing trees. We will go back later on the differences between their Lambek inspired grammars and our more abstract way of looking at the semantic parsing of a sentence. As the last papers using this view propose using a learned representation for words, we refer to Appendix B for a compositional view on these representations.

On a completely different approach, [7] provide a categorical structure based on Hopf algebra and coloured operads to explain their model of syntax. Their approach allowed to reconcile the works on syntactic and morphosyntactic trees in [13]. Similarly, [10] provides a modeling of CFGs using coloured operads. Our approach is based on the suggestion that merge in syntax can be done using labels, independent on how it is mathematically modelled and thus can be understood in both those frameworks (see Appendix C).

Of course, the works most related to this report are the ones on the notion of monadic types, see for example [1] or previous works by Simon Charlow.

2 Categorical Semantics of Effects: A Typing System

In this section presenting the way our typing system works, we will designate by $\mathcal{L}$ our language, as a set of words (with their semantics) and syntactic rules used 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. We refer to Appendix A for an example model of the English language. This is purely a formalization of the notions presented in [2].

2.1 Typing Category

2.1.1 Types

Let $\mathcal{C}$ be a closed cartesian category. This represents our main typing system, for words $\mathcal{O}(\mathcal{L})$ that can be expressed without effects. Remember that $\mathcal{C}$ contains a terminal object $\perp$ representing the empty or unit type. 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. This means our category objects are any abstract object that can be attained in a finite number from a finite number of functorial applications from an object of $\mathcal{C}$. $\bar{\mathcal{C}}$ will be our typing category.

We consider for our types the quotient set $\star = \mathrm{Obj}\left(\bar{\mathcal{C}}\right)/\mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ 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 following [9]). 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 $\perp \to \tau$ which we will denote directly by $\tau \in \star_0$.

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

We will provide in Figure 14 a list of the effect-less usual types associated to regular linguistic objects.

2.1.2 Functors, Applicatives and Monads

In our framework, we consider *language functors*[i] as polymorphic functions: for a set of base types S, a functor is a function:

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

[i] The elements of our language.

Since $\star$ is a fibration of the types in $\bar{\mathcal{C}}$, if a functor can be applied to a type, it can also be applied to all *affected* versions of that type, i.e. $\mathcal{F}(L)^*(\tau \in \star)$.

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 gives the following typing judgements:

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

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

In the same fashion, we can consider functions to have a type in $\star$ or more precisely of the form $\star \to \star$ 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:

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

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

2.1.3 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 $\text{Id}_{\mathcal{C}} \Rightarrow L \circ R$ and $R \circ L \Rightarrow \text{Id}_{\mathcal{C}}$. In particular we get a monad and a comonad from a canonical setting.

2. To deal with the resolution of effects, we can map handlers to natural transformations which go from some functor F to the Id functor, allowing for a sequential computation of the effects added to the meaning. We will develop a bit more on this idea in Paragraph 2.1.3.2 and in Section 3.

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 2.1.3.3.

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 defining the properties of a natural transformation provides a way to construct the *correct function* on the *correct functor* side of types.

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 [15]. This will guarantee for us that given a *Haskell* construction for a polymorphic function, we will get the associated natural transformation.

2.1.3.1 Adjunctions

We will not go in much details about adjunctions, as a full example and generalization process is provided in A.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 $\eta : \text{Id}_{\mathcal{A}} \Rightarrow R \circ L$ and $\varepsilon : L \circ R \Rightarrow \text{Id}_{\mathcal{A}}$ 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

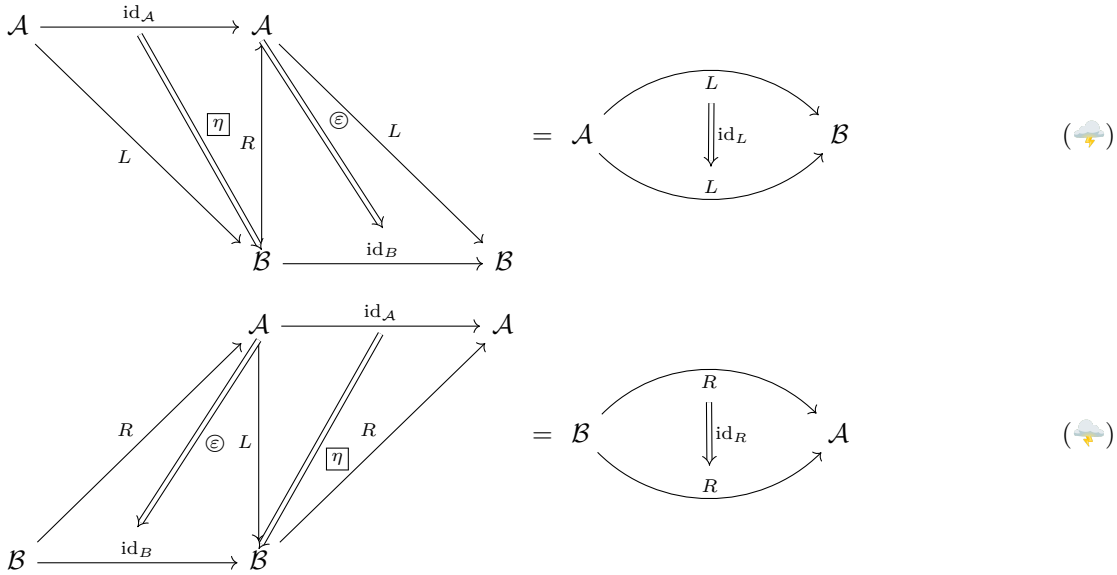


Figure 1: Zig-Zag (⛈) and Zag-Zig (⛈) equations defining adjunctions

a typing system which uses effects: adjunctions provide a way to combine effects and to handle them, allowing to simplify the computations on the free monoid on the set of functors.

2.1.3.2 Handlers

As introduced by [8], we use handlers to reduce effects by adding them to the syntactic tree. As considered by [17] and [14], handlers are to be seen as natural transformations describing the free monad on an algebraic effect. Considering handlers in this way allows us to directly handle computations inside the typing system. Using the framework proposed in [14] we simply need handlers for our effects/functors and we will then have in our language the constructs needed. The only thing we will require from an algebraic handler h is that for any applicative functor of unit η, $h \circ \eta = \mathrm{id}$.

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. This implies the choice of either one of the options is left during the implementation of the system.

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 ambiguity that arises from the phrase *A chair*. This usual example of the differences between the cognitive representation of words is actually a specific example of the different possible handlers for the powerset representation of non-determinism/indefinites: there are $|S|$ arrows from the initial object to S in *Set*, representing the different elements of S. In that sense, while handlers may have a normal form or representation purely dependant on the effect, the actual handler does not necessarily have a canonical form. This is the difference with the adjunctions: adjunctions are intrinsic properties of the coexistence of the effects, 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 modeling of handlers as natural transformations and most importantly, this does not add ambiguity to our model: The ambiguity that arises from the choice of possible handlers does not add to the ambiguity in the parsing.

2.1.3.3 Higher-Order Constructs

We might want to add plurals, superlatives, tenses, aspects and other similar constructs which act as function modifiers, inside our type system. For each of these, we give a functor Π corresponding to a new class of types along with natural transformations for all other func-

tors 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.

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 even 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**[ii]. 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):

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

This allows us to add complexity, not in the compositional aspects but inside the lexicon of our language, by constructing predicate modifiers passing down through a tree.

$$\begin{aligned}\mathbf{future}\,(\mathbf{be})\,(\mathbf{arg_1}, \mathbf{arg_2}) &\xrightarrow{\eta} \mathbf{future}\,(\mathbf{be})\,(\mathbf{arg_2})\,(\mathbf{future}\,(\mathbf{arg_1})) \\ &\xrightarrow{\eta} \mathbf{future}\,(\mathbf{be})\,(\mathbf{future}\,(\mathbf{arg_2}))\,(\mathbf{future}\,(\mathbf{arg_1}))\end{aligned}$$

2.1.3.4 Monad Transformers

In [2], 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.

2.2 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.

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

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

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

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

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

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

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

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

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

Figure 2: Typing and Subtyping Judgements

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.

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.

This non-determinism is a component of our language's grammar and semantics: a same sentence can have multiple interpretation without context. We will provide methods in Sections 3 and 4 to limit ambiguity to a minimum. 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 4 for more details.

Moreover, the observant reader might have noticed that our typing system is not decidable, because of the `nat/pure/return` rules which may allow for unbounded derivations. This is not actually an issue because of considerations on handling, as semantically void units will get removed at that time. This leads to derivations of sentences to be of bounded height, linear in the length of the sentence.

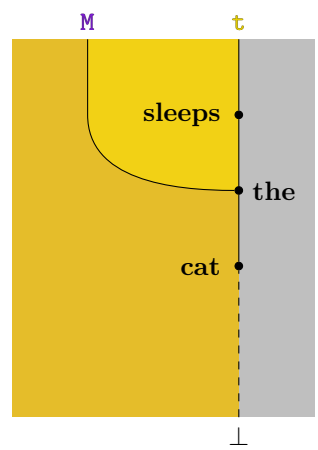
3 Handling Non-Determinism

The typing judgements proposed in Section 2.2 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.

3.1 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 the 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. 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 isotopic are equal [6] 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.



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

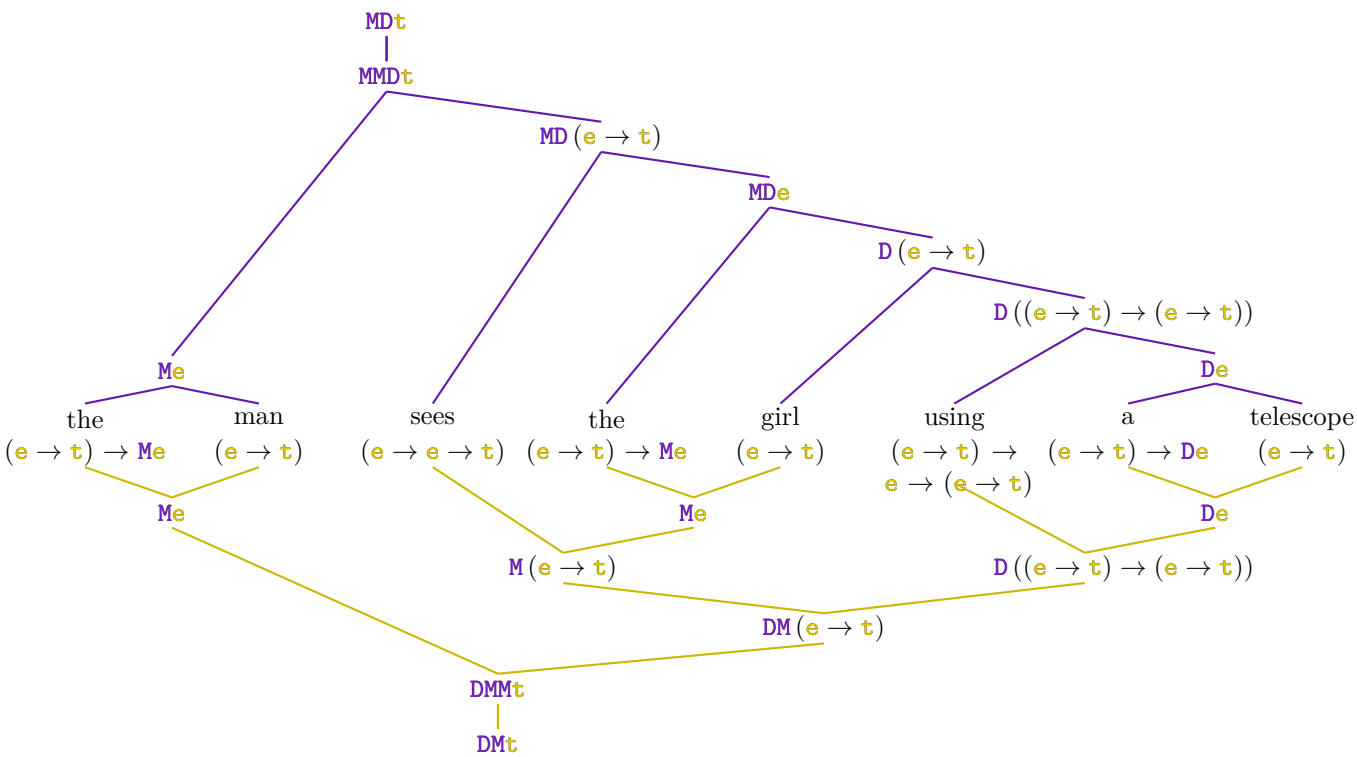
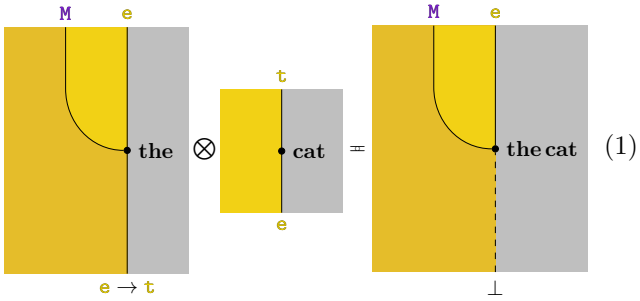


Figure 3: Parsing trees for the typing of *The man sees the girl using a telescope*.

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.

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. This will be discussed in more detail in Section 4.



To justify our proposition to only consider fully reduced expressions note that in this formalism we do not consider the expressions for our functors and natural transformations but simply the sequence in which they are applied.

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 artistic touch.

A previous modeling of parsers by string diagrams has already been proposed ([3], [16]). We will develop on the differences between those in Section 4.2.2.

3.2 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.

First, Theorem 3.1 reminds the main result by [6] about string diagrams which shows that our artis-

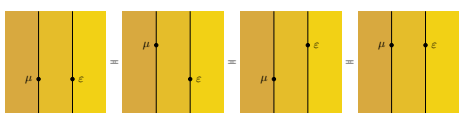
tic representation of diagrams is correct and does not modify the equation or the rule we are presenting.

Theorem 3.1 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.

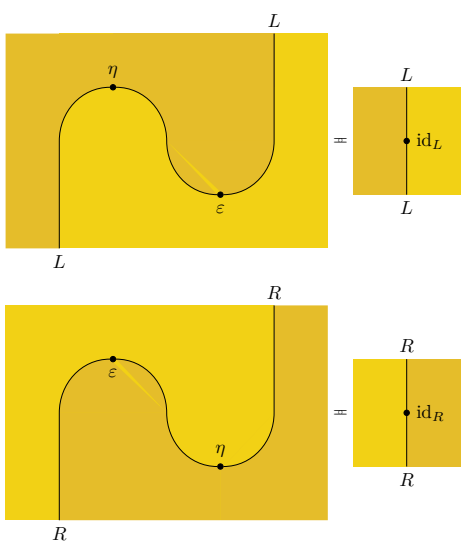
We will not prove this theorem here and refer to Theorem 3.1 in [12], and Theorem 1.2 in [6] for proofs.

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

The *Elevator* Equations are a consequence of Theorem 3.1 but also highlight one of the most important properties of string diagrams in their modeling of multi-threaded computations: what happens on one string does not influence what happens on another in the same time:

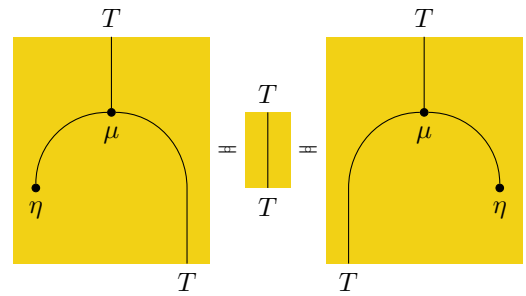


The *Snake* Equations are a rewriting of the categorical diagrams equations ( ) and ( ) of Paragraph 2.1.3.1 which are the defining properties of an adjunction. If we have an adjunction $L \dashv R$:

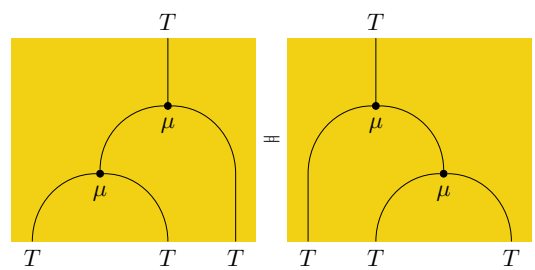


Note that we could express these equations using the following section, though it is, for now, a bit easier to keep it that way.

The *(co-)Monadic* Equations are a rewriting of the equations defining the associativity and the identity properties of a monad. Here we do not present the co-monadic equations which are the categorical dual of those ones.



(η)



(μ)

Considering the equivalence relation $\mathcal{R}$ freely generated from $\left\{ ( ), ( ), ( ), (\mu), (\eta) \right\}$ and the equivalence relationship $\mathcal{R}'$ of planar isotopy from Theorem 3.1, we get a set of normal forms $\mathcal{N}$ from the set of all well-formed parsing diagrams $\mathcal{D}$ defined by: $\mathcal{N} = (\mathcal{D}/\mathcal{R})/\mathcal{R}'$

3.3 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.

[4] proposed a combinatorial description to check in linear time for equality under Theorem 3.1. To provide with more flexibility 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.

3.3.1 Representing 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 $(D.N, D.S, D.L)$ 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 $nat = (nat.\mathrm{h}, nat.\mathrm{in}, nat.\mathrm{out})$. 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

Algorithm 1 Validity Check

```
function isValid(D)
    S ← D.S
    for i < D.N do
        nat ← D.L(i)
        b, e ← nat.h, nat.h + |nat.in|
        if S[b : e] ≠ nat.in then return False
        S[b : e] ← nat.out
    return True
```

all effects are handled for validity.

Our structure can be normalized in polynomial time to check for equality under Theorem 3.1 from [4]. More precisely, the complexity of our algorithm is in $\mathcal{O}\left(n \times \sup_i |D.\mathrm{L}(i).\mathrm{in}| + |D.\mathrm{L}(i).\mathrm{out}|\right)$, which depends on our lexicon but most of the times will be linear time, although this matters when considering Appendix B, which yields quadratic time.

3.3.2 Equational Reductions

We are faced a problem when computing reductions using the equations for our diagrams which is that by definition, an equality is symmetric. To solve this issue, we only use equations from left to right to reduce as much as possible our result instead.

Theorem 3.2 — 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.

The Snake Equations First, let us 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:

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

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

The Monadic Equations For the monadic equations, we only use Equation η as a way to reduce the number of natural transformations, since the goal here is to attain normal forms and not find all possible reductions. We ask that equation μ is always used in the direct sense $\mu(\mu(TT), T) \to \mu(T\mu(TT))$ so that the algorithm terminates. We use the same

convention for the comonadic equations. The validity conditions are as easy to define for the monadic equations as for the *snake* equations when considering diagrams in *right* normal forms. Then, for Equation (η) we simply delete the nodes and for Equation (μ) we switch the horizontal positions for i and $i+1$.

Proof of the Confluence Theorem. The first point of this theorem is exactly Theorem 4.2 in [4]. 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 3.2. ■

Theorem 3.3 — Normalization Complexity Reducing a diagram to its normal form is done in polynomial time in the number of natural transformations in it.

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

3.3.3 Completeness, Soundness, Decidability

Let us do a short 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 3.3, clearly, our system is decidable. 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.

4 Efficient Semantic Parsing

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

4.1 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 [2] provides a way to do so by constructing semantic trees labelled by sequence of *combinators* (see Section 4.2). 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 4 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.

This leads the induced algorithm (for computing the set of denotations) to be exponential in the input in the

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

Figure 4: List of possible combinations for different presuppositions for inputs, as a definition of a function PT from proof trees to proof trees.

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

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

$$\dfrac{\dfrac{}{\Gamma \vdash l : \mathtt{Fa} \to \mathtt{b}}\top \quad \dfrac{}{\Gamma \vdash r : \mathtt{Fa}}\top}{\Gamma \vdash lr : \mathtt{b}}\text{App}$$

Those two different proof trees yield different semantic interpretations which may both be useful, as discussed by [2], especially in their analysis of closure properties, which is modified in the following section for a better integration in the typing system.

4.1.1 Islands

[2] provide a formal analysis of islands[iii] based on islands being a different type of branching nodes in the syntactic tree of a sentence, which asks to resolve all $\mathtt{C}$ effects[iv] before that node or being resolved at that node.

To reconcile this directly 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, we *mark* each node inside the island by adding a "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 change the boundary word's type so it cannot take a continuation as input, 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 $\mathtt{C}$, handle it inside the other functors and keep going, on both sides, where PT is defined in Figure 4:

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

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

This idea amounts to seeing islands, whatever their type, as a form of grammatical/syntactic effect, which is still part of the language but not of the lexicon, a bit like future, modality or plurals, without the semantic component. The idea of seeing everything as effects, while semantically void, allows us to translate into our theory of type-driven semantics outer information from syntax, morphology or phonology that influences the semantics of the sentence, and make use of the type system to enforce them. Other examples of this can be found in the modeling of plural (for morphology, see Sections 2.1.3.3 and A.2.4) and the emphasis of the words by the $\mathtt{F}$ effect (for phonology), and show the empirical well-foundedness of this point of view. While we do not aim

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

[iv] Those represent quantifications.

to provide a theory of everything linguistics through categories, 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.

4.2 Syntactic-Semantic Parsing

4.2.1 The Improved Method

As using a naïve strategy on the trees results in an exponential algorithm, we will 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. Here, we change our notations from the proof trees of Figure 4 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 [2] as it simplifies the rewriting of our typing judgements in a CFG. The grammars provided in Figures 13 and 5 are the ones used in the parsing.

The grammar in Figure 5 works in five major sections:

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 [2].
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.

We do not prove here that these typing rules cannot be written in a simpler syntactic structure.

The idea of the *grammatical* reduction is that from two words that can syntactically combine, we use the binary effect combinators, before choosing the appropriate binary type combinator. It is at this point in the reduction we actually do the compositional part. We close up the reduction by possibly using unary effect combinators. This work is done for combinators of arity two, but could be extended, as seen in Appendix B.

$$
\begin{array}{lcl}
\tau & ::= & \tau \Rightarrow \tau \\
 & \mid & \langle \tau, \tau \rangle \\
 & \mid & \bar{\tau} \\
 & \mid & \mathrm{F}\tau \\
\\
\bar{\tau} & ::= & \mathtt{t} \mid \mathtt{e} \mid \cdots \\
\\
\mathrm{F} & ::= & \mathcal{F}\left(\mathcal{L}\right)^{*} \\
\\
\mathrm{Tree} & ::= & \tau, \tau
\end{array}
\qquad
\begin{array}{lcl}
\mathrm{SM} & ::= & \mathrm{ML} \mid \mathrm{MR} \\
 & \mid & \mathrm{UL} \mid \mathrm{UR} \\
 & \mid & \mathrm{A} \\
 & \mid & \mathrm{J} \\
 & \mid & \mathrm{C} \\
 & \mid & \mathrm{ER} \mid \mathrm{EL} \\
 & \mid & \mathrm{DN} \\
\\
\mathrm{M} & ::= & \mathrm{SM}, \mathrm{M} \\
 & \mid & \mathrm{BComb} \\
\\
\mathrm{BComb} & ::= & > \mid < \mid \wedge \mid \vee
\end{array}
$$

$$
\begin{array}{lcl}
>, \beta & ::= & (\alpha \to \beta), \alpha \\
<, \beta & ::= & \alpha, (\alpha \to \beta) \\
\wedge, \alpha \to \mathtt{t} & ::= & (\alpha \to \mathtt{t}), (\alpha \to \mathtt{t}) \\
\vee, \alpha \to \mathtt{t} & ::= & (\alpha \to \mathtt{t}), (\alpha \to \mathtt{t})
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{J}_{\mathtt{F}}\ \mathtt{F}\tau & ::= & \mathtt{FF}\tau \\
\mathrm{DN}_{\mathtt{C}}\ \tau & ::= & \mathtt{C}_{\tau}\tau
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{ML}_{\mathtt{F}}\left(\alpha, \beta\right) & ::= & \mathtt{F}\alpha, \beta \\
\mathrm{MR}_{\mathtt{F}}\left(\alpha, \beta\right) & ::= & \alpha, \mathtt{F}\beta \\
\mathrm{A}_{\mathtt{F}}\left(\alpha, \beta\right) & ::= & \mathtt{F}\alpha, \mathtt{F}\beta \\
\mathrm{UR}_{\mathtt{F}}\left(\alpha \to \alpha', \beta\right) & ::= & \mathtt{F}\alpha \to \alpha', \beta \\
\mathrm{UL}_{\mathtt{F}}\left(\alpha, \beta \to \beta'\right) & ::= & \alpha, \mathtt{F}\beta \to \beta' \\
\mathrm{C}_{\mathtt{LR}}\left(\mathtt{L}\alpha, \mathtt{R}\beta\right) & ::= & (\alpha, \beta) \\
\mathrm{ER}_{\mathtt{R}}\left(\mathtt{R}\left(\alpha \to \alpha'\right), \beta\right) & ::= & \alpha \to \mathtt{R}\alpha', \beta \\
\mathrm{EL}_{\mathtt{R}}\left(\alpha, \mathtt{R}\left(\beta \to \beta'\right)\right) & ::= & \alpha, \beta \to \mathtt{R}\beta'
\end{array}
$$

Figure 5: Possible Type Combinations in the form of a near CFG

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

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

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

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

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

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

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

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

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

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

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

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

Figure 6: Denotations describing the effect of the combinators used in the grammar describing our combination modes presented in Figure 5

For example, the rules of the form $\mathrm{ML}_{\mathtt{F}}M,\tau' \leftarrow M,\tau$ are rules that provide ways to combine effects from the two inputs in the order we want to: we can combine $\mathtt{RSe}$ and $\mathtt{CW}(\mathtt{e} \rightarrow \mathtt{t})$ into $\mathtt{RCWSt}$ with the mode MLMRMRML $<$ (see [2] 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 of the combinator, and are described in Figure 6. The main reason we need to get denotations associated to combinators, is to properly define the combination of words/phrases that underlines our system. The important thing on those denotations is that they're a direct translation of the rules defining the notions of functors, applicatives, monads and thus are not specific to any denotation system, even though we will use lambda-calculus styled denotations to describe them. This makes us able to compute the actual denotations associated to a sentence using our formalism, as presented in figure 7. 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 their denotations work, for simplicity of writing the denotations, and simplicity of doing the parsing.

Since our grammar is defined with respect to the functor set, the algorithm which parses the combinators will have a complexity in the size of $\mathcal{F}(\mathcal{L})$ that is linear, which comes from the fact that our grammar's size is linear in $|\mathcal{F}(\mathcal{L})|$.

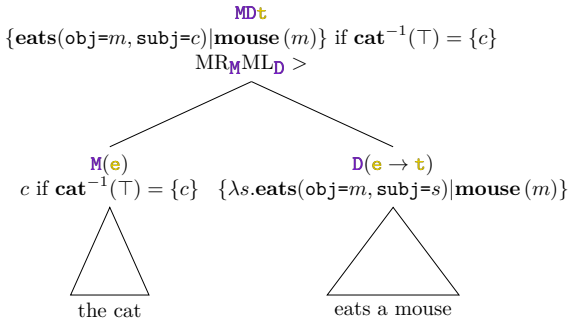
Theorem 4.1 Semantic parsing of a sentence is polynomial in the length of the sentence and the size of the type system and syntax system.

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 $|G_\tau|$. ■

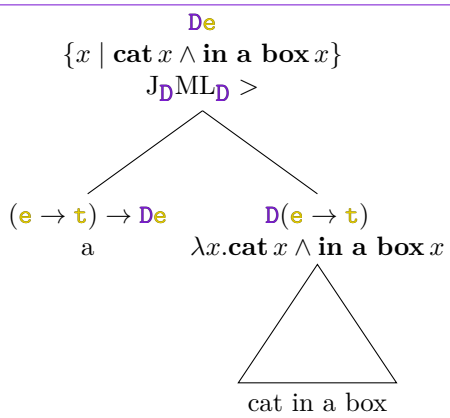
While we have gone with the hypothesis that we have a CFG for our language, any type of polynomial-time structure could work, as long as it is as expressive as a CFG. We will do the following analysis using a CFG since it allows to cover enough of the language for our example and for simplicity of describing the process of adding the typing CFG, even though some think that English is not a context free language [5].

Theorem 4.2 Retrieving a pure denotation for a sentence can be done in polynomial time in the length of the sentence, given a polynomial time syntactic parsing algorithm and polynomial time combinators.

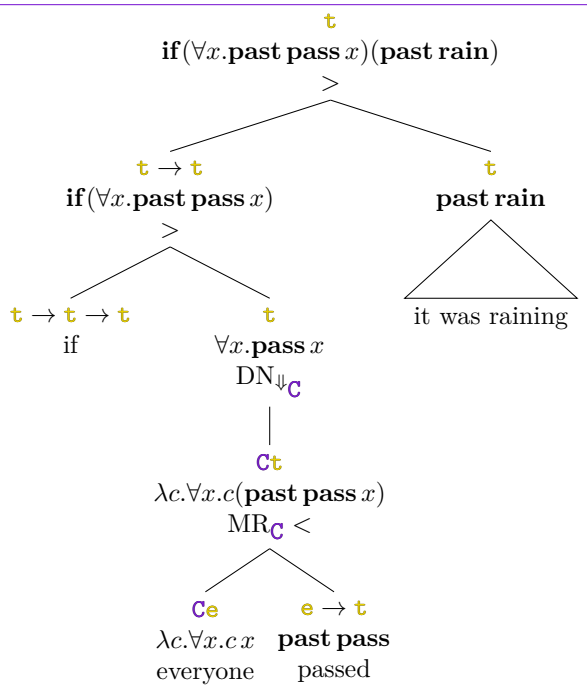
Proof. We have proved in Theorem 4.1 that we can retrieve a semantic parse tree from a sentence in polynomial time in the input. Since we also have shown that the length of a semantic parse tree is quadratic in the length of the sentence it represents, being linear in the length of a syntactic parse tree linear in the length of



(a) Labelled tree representing the equivalent parsing diagram to 9



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



(c) Labelled tree representing the equivalent parsing diagram to 12

Figure 7: Examples of Labeled Parse Trees for a few sentences.

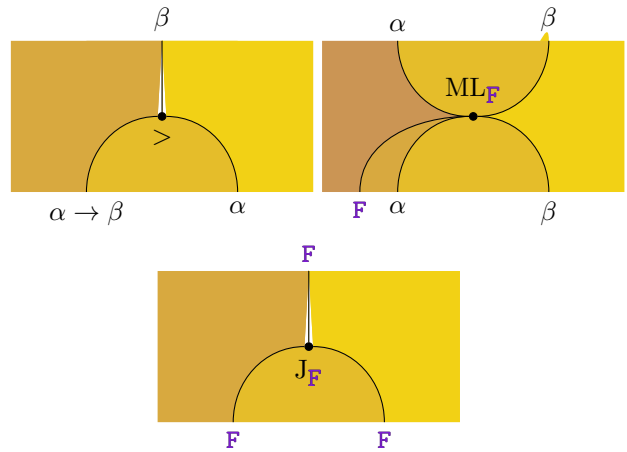


Figure 8: String Diagrammatic Representation of Combinator Modes $>$, ML and J

the sentence. We have already seen that given a denotation, 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 algorithm 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 A, 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[v] for the same reason.

4.2.2 Diagrammatical Parsing

When considering [3] 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 see our combinators as natural transformations (2-cells): then we can see the different sets of combinators as different arity natural transformations. $>$, ML$_{\mathtt{F}}$ and J$_{\mathtt{F}}$ are represented in Figure 8, up to the coloring of the regions, because that is purely for an artistic rendition.

Now, a way to see this would be to think of this as an orthogonal set of diagrams to the ones of Section 3: we can use the syntactic version of the diagrams to model our parsing, according to the rules in Figure 5, and then

[v] Depending on the functor but still bounded by a constant.

combine the diagrams as shown in Figure 9. This explains the construction of the diagrams in Section 3. In this figure we exactly see the sequence 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 operations on the strings, and don't actually allow for braiding inside the diagrammatic calculus. To better understand what happens in those parsing diagrams, Figure 7 provides the translations in labelled trees of the parsing diagrams of Figures 9, 10 and 12.

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 [3], as it is seemingly a typed version, where we change the Pregroup category for the typing category, possibly taken with a product for representation of the English grammar representation, to accomodate for syntactic typing on top of semantic typing. This is, again, just another rewriting of our typing rules.

We have a first axis of string diagrams in the category $\mathcal{C}$ - our string diagrams for effect handling, as in Section 3 - and a second *orthogonal* axis of string diagrams on a larger category, with endofunctors graded by the types in our typing category $\bar{\mathcal{C}}$ and with natural transformations mapping the combinators defined in Figures 5 and 6. 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 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. These diagrams are an extension of the parse trees presented in [2] in a graphical format and extended to be integrated

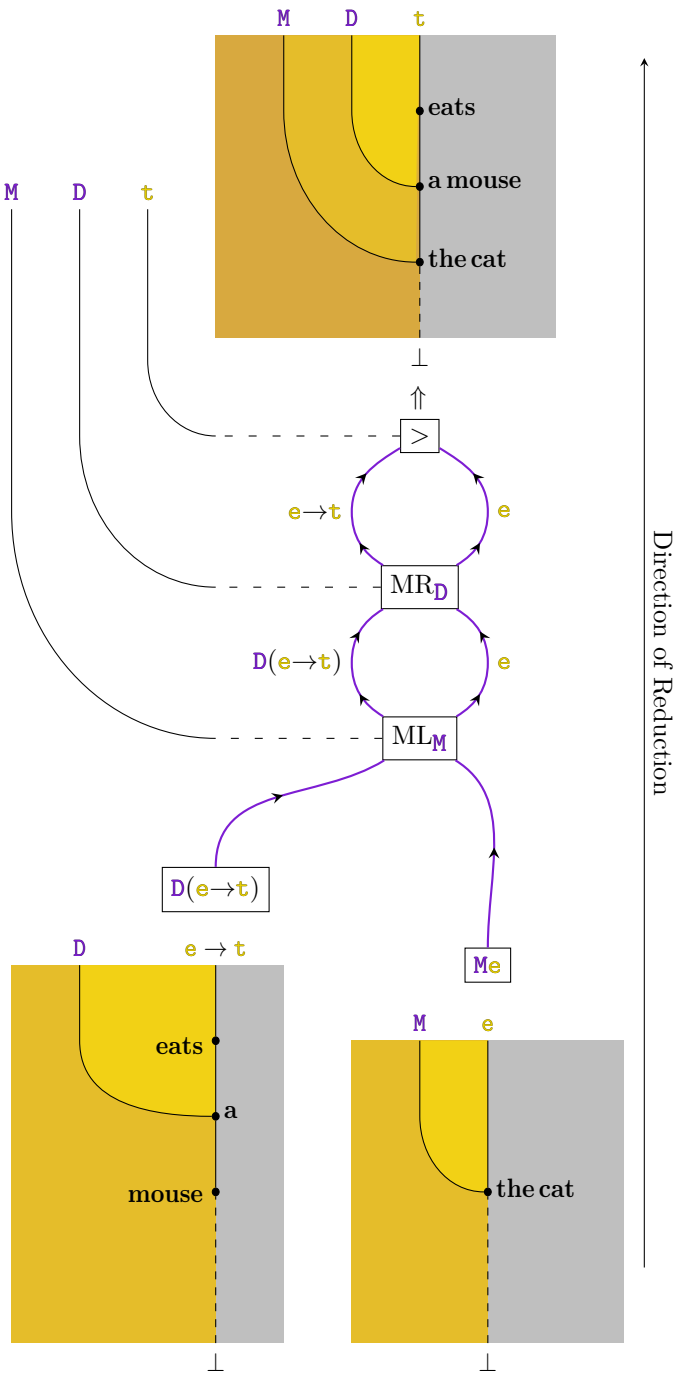


Figure 9: Representation of a parsing diagram for the sentence *the cat eats a mouse*. See Figure 7b for translation in a parse tree.

with the handling of effects, as described in Section 3, forming a full diagrammatic calculus system for semantic parsing.

Applying the unary combinators which reduce effects to the actual *parsing* part of the diagram is done in the exact same way as in the CFG, we apply them as needed, and they 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 10. 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, which justifies again the shape of the diagrams in Section 3.

One of the main reasons why this point of view of diagrammatic 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[vi] 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 12. The sections in the rectangle represent what happen when considering our combination step as implementing patterns inside a knitwork, as seen in Figure 11. The dif-

[vi] 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.

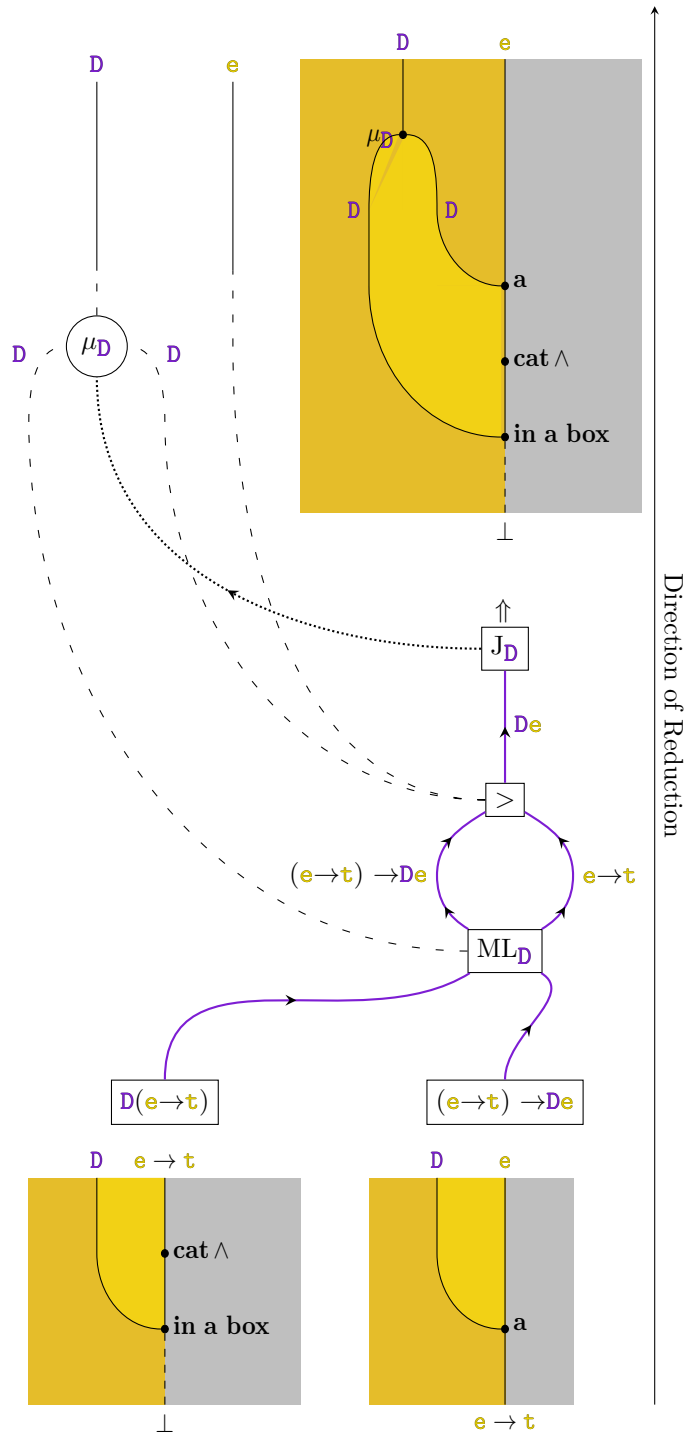


Figure 10: Example of a parsing diagram for the phrase *a cat in a box*, presenting the integration of unary combinators inside the connector line. See Figure 7b for translation in a parse tree.

Figure 11: Example of a *Jacquard* knitwork. Photography and work courtesy of the author's mother.

ferent 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.

4.3 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.

First, let's consider the reductions proposed in Section 3. Those define normal forms under Theorem 3.2 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. 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:

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

Indeed, in that case we could either get a result with effects $\mathtt{FG}$ or with effects $\mathtt{GF}$. 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. 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:

- $m_{\mathtt{F}}, \mathrm{DN}, m_{\mathtt{F}}$ where $m \in \{\mathrm{MR}, \mathrm{ML}\}$
- $\mathrm{ML}_{\mathtt{F}}, \mathrm{DN}, \mathrm{MR}_{\mathtt{F}}$
- $\mathrm{A}_{\mathtt{F}}, \mathrm{DN}, \mathrm{MR}_{\mathtt{F}}$
- $\mathrm{ML}_{\mathtt{F}}, \mathrm{DN}, \mathrm{A}_{\mathtt{F}}$
- C

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

- $\{m_{\mathtt{F}}, j, m_{\mathtt{F}}\}$ where $m \in \{\mathrm{MR}, \mathrm{ML}\}$
- $\mathrm{ML}_{\mathtt{F}}, j, \mathrm{MR}_{\mathtt{f}}$
- $\mathrm{A}_{\mathtt{F}}, j, \mathrm{MR}_{\mathtt{F}}$,
- $\mathrm{ML}_{\mathtt{F}}, j, \mathrm{A}_{\mathtt{F}}$
- k, C for $k \in \{\varepsilon, \mathrm{A}_{\mathtt{F}}\}$
- If $\mathtt{F}$ is commutative as a monad:
  - $\mathrm{MR}_{\mathtt{F}}, \mathrm{A}_{\mathtt{F}}$
  - $\mathrm{A}_{\mathtt{F}}, \mathrm{ML}_{\mathtt{F}}$
  - $\mathrm{MR}_{\mathtt{F}}, j, \mathrm{ML}_{\mathtt{F}}$
  - $\mathrm{A}_{\mathtt{F}}, j, \mathrm{A}_{\mathtt{F}}$

Similar facts can be given for the co-units of an adjunction.

Theorem 4.3 The rules proposed above yield equivalent results.

Proof. For the first point, the equivalence has already been proved under Theorem 3.2. 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[vii] is to at some point have something get in between the two.

[vii] Provided they can be joined.

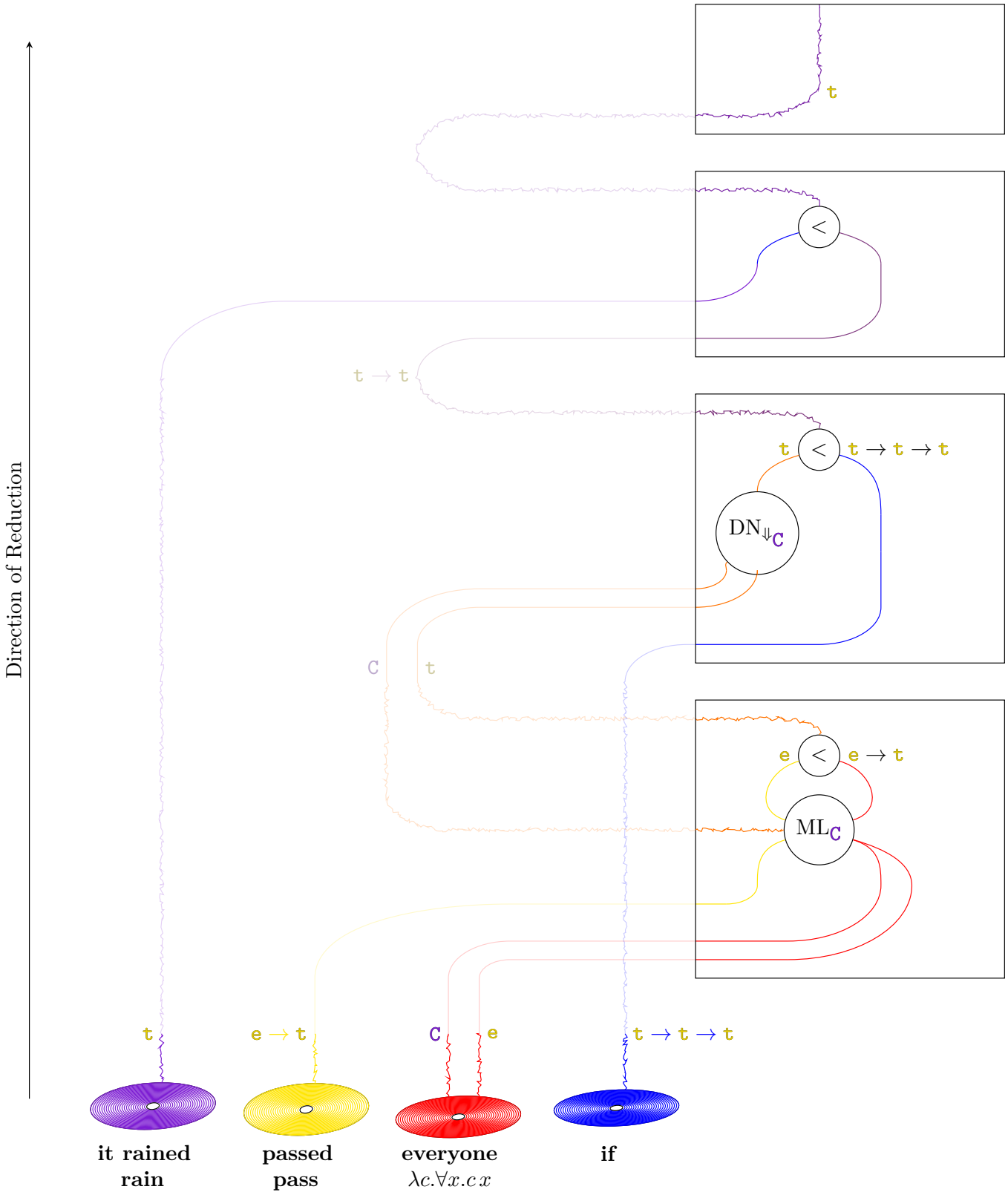


Figure 12: 3D-like Representation of the Diagrammatic Parsing of a Sentence. See 7c for the translation in a Parse Tree

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 3.1. 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 facts, as we should use the units of monads over applicative rules and fmap. ■

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*). We are far from reducing to a minimum the number of different paths possible to get a same final denotation, but combining the different reduction schemes for syntax, effect-handling and denotations in a larger scheme will be a step in the right direction.

When using our diagrammatic approach to parsing, we can write all the reductions described above to our paradigm as in Section 3: it amounts to constructing a set of normal forms for the string diagrams. This leads to the same algorithms developed in Section 3 being usable here: we just have a new improved version of Theorem 3.2 which adds the normal forms specified in this section to the newly added *syntactic* axis of diagrammatic computations. 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.

As before, there is no evidence that our system is complete, if not the contrary, so the arguments developed in Section 3.3.3 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.

5 Bibliography

[1] A. Asudeh and G. Giorgolo. *Enriched Meanings: Natural Language Semantics with Category Theory*. Oxford Studies in Semantics and Pragmatics. Oxford University Press, Oxford, New York, Nov. 2020.

[2] D. Bumford and S. Charlow. Effect-driven interpretation: Functors for natural language composition, Mar. 2025.

[3] B. Coecke, M. Sadrzadeh, and S. Clark. Mathematical Foundations for a Compositional Distributional Model of Meaning, Mar. 2010.

[4] A. Delpeuch and J. Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm, Jan. 2022.

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

[6] A. Joyal and R. Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, July 1991.

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

[8] J. Maršík and M. Amblard. Algebraic Effects and Handlers in Natural Language Interpretation.

[9] P.-A. Melliès and N. Zeilberger. 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, Jan. 2015. ACM.

[10] P.-A. Melliès and N. Zeilberger. The categorical contours of the Chomsky-Sch\"{u}tzenberger representation theorem. *Logical Methods in Computer Science*, Volume 21, Issue 2:13654, May 2025.

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

[12] P. Selinger. A survey of graphical languages for monoidal categories. volume 813, pages 289–355. 2010.

[13] I. Senturia and M. Marcolli. The Algebraic Structure of Morphosyntax, June 2025.

[14] B. van den Berg and T. Schrijvers. A framework for higher-order effects & handlers. *Science of Computer Programming*, 234:103086, May 2024.

[15] P. Wadler. 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, Nov. 1989. Association for Computing Machinery.

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

[17] N. Wu, T. Schrijvers, and R. Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA, Sept. 2014. Association for Computing Machinery.

```
CP    ::= DP, VP
        | Cmp, CP
        | CP, CBar

CBar  ::= Cor, CP

Dbar  ::= Cor, DP

DP    ::= DP, Dbar
        | Dmp, DP
        | Det, NP
        | Gen, TN

Gen   ::= DP, GenD

NP    ::= AdjP, NP
        | NP, AdjP

AdjP  ::= TAdj, DP
        | Deg, AdjP

VP    ::= TV, DP
        | AV, CP
        | VP, AdvP

TV    ::= DV, DP

AdvP  ::= TAdv, DP
```

Figure 13: Partial CFG for the English Language

A 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. Apart from Section A.2.4, the contents from this section are a rewriting from [2].

A.1 Syntax

In Figure 13 we provide a toy Context-Free Grammar to support the claims made in Section 4.2. The discussion 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 14. 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`

A.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.

S $\mathtt{t}$

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

ProperN $\mathtt{e}$

NP Either:

- $\mathtt{e}$ *John, a cat*
- $(\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$ *the cat, every man*

DET $\Upsilon(\mathbf{CN}) \to \Upsilon(\mathbf{NP})$

ADJ(P) Either:

- $(\mathtt{e} \to \mathtt{t})$ *carnivorous, happy*
- $(\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t})$ *skillful*

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

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

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

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

Figure 14: Usual Typings for some Syntactic Categories

A.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[viii], the types that are generally used are the following, and we will see it matches with our lexicon, and simplifies our functorial definitions[ix]. Those are based on [11]. Here Υ is the operator which retrieves the type from a certain syntactic category.

A.2.2 Lexicon: Semantic Denotations for Words

Many words will have basically the same "high-level" denotation. For example, the denotation for most common nouns will be of the form: $\Gamma \vdash \lambda x.\mathbf{planet}\, x : \mathtt{e} \to \mathtt{t}$. In Figure 15 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[x], 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 de-

[viii] Those are a subset of the ones proposed in Section A.1.
[ix] We don't consider effects in the given typings.
[x] Emphasis, for example.

fined in Figure 16 along with their respective `fmap`. For denotations that might not be clear at first glance, $\cdot_{\mathtt{F}}$ is the phonologic accentuation or emphasis of a word and $\cdot, \mathbf{a}\cdot$ 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.

A.2.3 Effects of the Language

For the applicatives/monads in Figure 16 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: $\mathtt{G}$ designates reading from a certain environment of type $\mathtt{r}$ while $\mathtt{W}$ encodes the possibility of logging a message of type $\mathtt{t}$ along with the expression. The functor $\mathtt{M}$ describes the possibility for a computation to fail, for example when retrieving a value that does not exist (see **the**). The $\mathtt{S}$ functor represents the space of possibilities that arises from a non-deterministic computation (see **which**).

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

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

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 $\mathtt{W} \circ \mathtt{G} \Rightarrow \mathrm{Id}$.

A.2.4 Modality, Plurality, Aspect, Tense

We will now explain ways to formalise higher-order concepts as described in Section 2.1.3.3.

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 modeling makes sense. As such, we give ourselves a way to measure if a certain x is plural our not, which we will denote by $|x| \geq 2$[xi]. It is important to note that if τ is a type, and $\Gamma \vdash x : \tau$ then we should have $|\Pi x| \geq 1 = \top$[xii]. 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[xiii]. This one depends on the *syntactic*[xiv] type of the arrow, as seen in Figure 15. 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 2.1.3.3. To understand a bit more why this works as described in our natural transformation rules, consider $\Pi\,(\mathbf{which})$.

[xi] 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.

[xii] This might be seen as a typing judgement !

[xiii] `fmap`, basically.

[xiv] Actually it depends on the word considered, but since the denotations (when considered effect-less) provided in Figure 15 are more or less related to the syntactic category of the word, our approximation suffices.

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

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
$\mathtt{T}(\tau) = \mathtt{s} \to (\tau \times \mathtt{s})$	$\mathtt{D}\varphi(\lambda s.\,\{\langle x, x +\!\!+ s\rangle \mid px\}) = \lambda s.\,\langle \varphi x, \varphi x +\!\!+ s\rangle$	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
$\mathtt{D}(\tau) = \mathtt{s} \to \mathtt{S}(\tau \times \mathtt{s})$	$\mathtt{D}\varphi(\lambda s.\,\{\langle x, x +\!\!+ s\rangle \mid px\}) = \lambda s.\,\{\langle \varphi x, \varphi x +\!\!+ s\rangle \mid px\}$	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 16: 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 17: (Partial) Definition for the Π Plural Functor

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:

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

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

Now, in quite a similar way, we can create functors from any sort of judgement that can be seen as a function $\mathtt{e} \to \mathtt{t}$ in our language[xv]. 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 Multiple Ways

In this section we will discuss what can happen when the denotation system we consider is not actually based on a compositional model, or is independent of the syntactic model. In that case, our model of using typing to retrieve parsing trees will need modifications to be integrated.

In the case where the compositional aspects of the sentence are blurred by a non-binary arity, whether it's variable (leading to non-binary trees) or simply non-compositional and makes use of all the words of the sentence (or the ones prior to the last one read, like in a LLM), we can make use of higher arity base combinators. We replace the function application by the combinator adapted to our model, then a theoretical (or learned) analysis of the concepts that can be modeled by adding effects can still be done, and our work can be completed in the way described above.

Of course in the case where no syntactic structure is used for semantics, using semantics to derive syntax is useless, but the formalism of string diagrams used to present the actual parsing can still be useful: it could also be used to model the attention patterns (as side-effects of the reading of a new token and adding its meaning to the current representation of the sentence). This is an insight of how to implement our enhancement onto a non-compositional system, although that of course removes a bit of the interest of the system, as typing will not matter in that case.

C Coproduct Integration

In this section based on the work by [7] and [13], 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

[xv] 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.

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 [2], 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 [10] 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 6 and Figure 12 in [2], or by looking at the parse trees equivalent to string diagrams in Figures 9, 10 and 12, 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[xvi] 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.

[xvi] Or more, see B