A Diagrammatic Calculus for a Functional Model of Natural Language Semantics

Anonymous author
Anonymous affiliation

Abstract

In this paper, we study a functional programming approach to natural language semantics, allowing us to increase the expressiveness of a more traditional denotation style. We will formalize a category based type and effect system to represent the semantic difference between syntactically equivalent expressions. We then construct a diagrammatic calculus to model parsing and handling of effects, providing a method to efficiently compute the denotations for sentences.

2012 ACM Subject Classification Models of computation. Document management and text processing.

Keywords and phrases Natural Language Semantics,Parsing,Side Effects,String Diagrams,Type System,Functional Programming

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Category Student Paper

Acknowledgements Anonymous acknowledgements

1 Introduction

What is *a chair*? How do I know that *Jupiter, a planet*, is *a planet*? To answer those questions, [1] provide 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 serving it.

Our formalism is inscribed in the contemporary natural language semantic theories which are 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 the reader to [1] to get an overview of the linguistic considerations that are the base of the theory.

In this paper, we will provide a formal definition of an enhanced type and effect system for natural language semantics, based on 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 potential 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.

2 Related Work

This is not the first time a categorical representation of compositional semantics of natural language is proposed, [2] already suggested an approach based on monoidal categories using an external model of meaning. What our approach gives more, is additional latitude for the definition of denotations in the lexicon, and a visual explanation of the difference between multiple possible parsing trees. The proposition of [10] is closer to our proposition on graphical aspects, but still has the limits of using an external model of meaning while ours expands on the use of an expanded model of computation. We will go back later on our more abstract way of looking at the semantic parsing of a sentence.

On a completely different approach, [5] provide a categorical structure based on Hopf algebra and coloured operads to explain their model of syntax, leading to results at the interface of syntax and morphology presented in [9]. Similarly, [8] 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.

3 Categorical Semantics of Effects: A Typing System

In this section, we will formalize a type system underlying the theory proposed in [1]. To do so, we will designate by $\mathcal{L}$ our language, as a set of words (with their associated meaning/denotation) and syntactic rules underlying the semantic combination. The absence of syntactic rules is allowed, although it partly defeats the purpose of this work. This might be useful when proposing compositional models of learned representations.

We will use $\mathcal{F}(\mathcal{L})$ to denote the set of functors or higher-order functions used in denotations of $\mathcal{L}$. Those are chosen when representing the language (see Figure 11b for examples), and should be additions to a simpler semantic theory. 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.

3.1 Typing Category

3.1.1 Types

Let $\mathcal{C}$ be a closed cartesian category representing the domain of types for the domains and co-domains of uneffectful denotations. $\mathcal{C}$ is our *main* typing system, consisting of types for words that can be expressed without effects (see Figure 11a for an example). The terminal object $\perp$ of $\mathcal{C}$ represents the empty type or the lack thereof. We consider as our typing category $\bar{\mathcal{C}}$ the categorical closure for exponentials and products of $\mathcal{F}(\mathcal{L})^{*}(\mathcal{C})$, which consists of all the different type constructors (ergo, functors) that could be formed in the language. In that setting our types are those that can be attained from a finite number of functorial applications from an object of $\mathcal{C}$.

Since $\mathcal{F}(\mathcal{L})$ only induces a preorder on $\mathrm{Obj}\left(\bar{\mathcal{C}}\right)$, we consider the relation on types $x \succeq y \Leftrightarrow \exists F, y = F(x)$ (which should be seen as a subtyping relation as proposed in [7]). We then consider for our types the quotient set $\star = \mathrm{Obj}\left(\bar{\mathcal{C}}\right) /$ where is the transitive closure of the subtyping relationship induced by functorial application. We also define $\star_0$ to be the subset of types containing only uneffectful types, i.e. $\mathrm{Obj}(\mathcal{C})$. In contexts of polymorphism, we identify $\star_0$ to the adequate subset of $\star$. In this paradigm, constant objects (or results of fully handled computations) are functions with type $\bot \to \tau$ which we will denote directly by $\tau \in \star_0$. This will be useful when defining base combinators in Section 5.

3.1.2 Functors, Applicatives and Monads

Our point of view has us consider *language functors*[1] as polymorphic functions: for a (possibly restrained) set of base types S, a functor is seen as a function:

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

This means that if a functor can be applied to a type, it can also be applied to all *affected* versions of that type, i.e. $\mathcal{F}(L)(\tau \in \star)$. This gives us two typing judgements for the functor F:

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

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. Examples of this difference are to be found in Figures 11a and 11b.

In this regard, 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 transformation* judgement allowing derivation of $M\tau$ from$MM\tau$. This is a special case of the natural transformation rule that we define in the next section.

3.1.3 Natural Transformations

We could add judgements directly for adjunctions and monads, but we generalize by adding judgements for natural transformations, as adjunctions and monadic rules are natural transformations which arise from *natural* settings. While in general we do not want to create natural transformations, we want to be able to express these in three situations:

1. Adjunctions, Monads and Co-Monads[2].
2. To deal with the resolution of effects as explained in Section 4
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 Section 3.1.3.2.

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. From a linguistic point of view, natural transformations allow us to reason directly about type coercions and their coherence in the typing system, whether that

[1] Words with denotations in $\mathcal{F}(\mathcal{L})$ which represent denotationally effectful constructions, e.g. "a" or "the". They are to be considered with opposition to the *type functors* which are the mathematical construct in $\mathcal{F}(\mathcal{L})$.
[2] Which are actually the same thing.

is transporting effects across functors as in Section 3.1.3.2 or collapsing nested effects and more generally handling them as presented in Section 3.1.3.1 and 4.

In the Haskell programming language, any polymorphic function is a natural transformation from the first type constructor to the second type constructor, as proved in [11]. This will guarantee for us that given a *Haskell* construction for a polymorphic function, we will get the associated natural transformation.

3.1.3.1 Handlers

As introduced by [6], the notion of handlers is to be considered as the way to solve effects that obfuscate the result of a computation. Following [12], we understand handlers as natural transformations describing the resolution of an algebraic effect: they are natural transformations from the effect to the identity functor, effectively resolving them. Considering handlers this way allows us to directly handle our computations inside our typing system and in particular inside our parsing algorithm. This process will mostly be described in Sections 4 and 5.

To define a handler h, we will only require that for any applicative functor of unit η, $h \circ \eta = \mathrm{id}$. This solves the issue of non-termination of the system. 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 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 as two different speakers may have different ways to resolve the non-determinism effect that arises from the phrase *A chair*. This is the difference with the adjunctions: adjunctions are intrinsic properties of the coexistence of the effects, while the handlers are user-defined.

3.1.3.2 Higher-Order Constructs

Functors may also be used 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 allows us to add complexity not in the compositional aspects but in the model of the language, by simply saying that those constructs are predicate modifiers passed down (with or without side effects) to the arguments of predicates:

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

Among other higher-order constructs that might be represented using effects are scope islands, which could be modelled by a functor that cannot be passed as argument to words that would otherwise need a closure to be applied first. See Figure 5c for an example, based on theory presented in [1], Section 5.4.

The term "*higher-order construct*" comes from the idea that those constructs are not generated by words but at the scale of the sentence, or even the syntax in the case of *scope islands*. As such, we will say that this type of functors are *external* to the lexicon.

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

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

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

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

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

Figure 1 Typing and subtyping judgements for implementation of effects in the type system.

3.2 Typing Judgements

To complete this section, Figure 1 gives a simple list of different typing composition judgements through which we also re-derive the subtyping judgement to allow for its implementation. Note that here, the syntax is not taken into account: a function is always written left of its arguments, whether or not they are actually in that order in the sentence.

Using these typing rules for our semantic parsing steps, it is important to see that our grammar will still bear ambiguity. The next sections will explain how to reduce this ambiguity in short enough time.

Moreover, our current 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 the considerations on handling, as semantically void units will get removed at that time. Indeed, from the property of handlers adding a unit and not modifying the effect before it is handled does not change anything to the result and will be removed. This leads to derivations of sentences to be of bounded height, linear in the length of the sentence.

4 Handling Ambiguity

The typing judgements proposed in Section 3.2 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.

4.1 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. 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 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.

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. This delimits our usage of string diagrams as ways to look at computations and a tool to provide equality rules to reduce ambiguity.

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 transformation between two such functors τ_1, τ_2 is exactly an arrow in $\mathcal{C}$ of type $\tau_1 \to \tau_2$. 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.

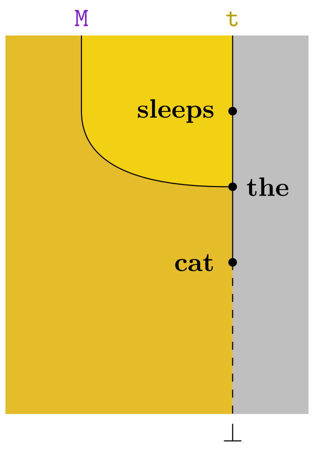


Figure 2 String diagram for the sentence *the cat sleeps*.

In the diagram of Figure 2, each string corresponds to a functorial effect or type layer applied during parsing. The base type string `t` is at the border of the gray area and is the one of the uneffectful denotation in $\mathcal{C}$ while the *functorial* string for `M` introduces the effect for optionality and possible failure of the computation. The question of providing rules to compose the string diagrams for parts of the sentences will be discussed in the next section, as it is related to parsing.

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.

4.2 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.

First, Theorem 1 reminds the main result by [4] 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 1** (Theorem 1.2 [4]). *String diagrams equivalent under planar isotopy in the graphical language are equal.*

A few equations on string diagrams also arise from the commutation of certain class of diagrams and thus typing judgements. We consider the *snake* equations are a rewriting of the categorical diagrams which are the defining properties of an adjunction and the *(co-)monadic* equations are the string diagrammatic translation of the properties of unitality and associativity of monads. These equations (and the reduction rules from Section 5.3) explain all the different reductions that can be made to limit non-determinism in our parsing and handling strategies.

4.3 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. In this section we formally describe the data structure we propose, as well as proving our system of rewriting allows us to compute normal forms for our diagrams.

4.3.1 Representing String Diagrams

We follow [3] in their combinatorial description of string diagrams. We describe a diagram by an ordered set of its 2-cells (the natural transformations, including handlers of the diagram) along with the number of input strings, for each 2-cell we log the following information:

- Its horizontal position: the number of strings that are right of it.
- Its type: an array of effects that are the inputs to the natural transformation and an array of effects that are the outputs to the natural transformation.

We will then write a diagram D as a tuple of 3 elements: $(D.N, D.S, D.L)$ 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 $nat = (nat.\mathrm{h}, nat.\mathrm{in}, nat.\mathrm{out})$. This gives a naïve algorithm in polynomial time to check if a string diagram is valid or not.

Because our representation contains strictly more information (without slowing access by a non-constant factor) than the one it is based on, our data structure supports the linear and polynomial time algorithms proposed with the structure by [3]. In particular our structure can be normalized in time $\mathcal{O}\left(n \times \sup_i |D.\mathrm{L}\left(i\right).\mathrm{in}| + |D.\mathrm{L}\left(i\right).\mathrm{out}|\right)$, which depends on our lexicon but most of the times will be linear time.

4.3.2 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. 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** (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 reduction rules for the different equations for our description of string diagrams.

The Snake Equations First, let's see when we can apply the equation for id_L 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 the equation at i if, and only if:

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

- $D.\mathrm{L}\,(i+1) = \varepsilon_{L,R}$
  This comes from the fact that we can't send either ε above η using right reductions and that there cannot be any natural transformations between the two. Obviously the equation for id_R works the same. 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 the unitality equation 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 associativity 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 unitality we simply delete the nodes and for associativity 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 [3]. 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 equation and the fact the other equations delete nodes. 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. ■

▶ **Theorem 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'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 *right* normalizing in linear time before to ensure we get all equational reductions and after to complete the reduction is enough, we have a polynomial time algorithm. ■

5 Efficient Semantic Parsing

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

5.1 Syntactic-Semantic Parsing

Using a naïve strategy of type checking on syntax trees yields an exponential algorithm. To avoid that, we extend the grammar system used to do the syntactic part of the parsing to include semantic combination of words. In this section, we will take the example of a CFG since it suffices to create our typing combinators, In Figure 3, we explicit a grammar of combination modes, based on [1] as it simplifies the rewriting of our typing judgements in a CFG.

This grammar works in five major sections:

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

$$
\begin{array}{lcl}
>, b & ::= & (a \to b), a \\
<, b & ::= & a, (a \to b) \\
\wedge, a \to \mathtt{t} & ::= & (a \to \mathtt{t}), (a \to \mathtt{t}) \\
\vee, a \to \mathtt{t} & ::= & (a \to \mathtt{t}), (a \to \mathtt{t}) \\
\\
\mathrm{J}_{\mathtt{F}}\ \mathtt{F}\tau & ::= & \mathtt{FF}\tau \\
\mathrm{DN}_{\mathtt{C}}\ \tau & ::= & \mathtt{C}_\tau \tau
\end{array}
\qquad
\begin{array}{lcl}
\mathrm{ML}_{\mathtt{F}}(\alpha, \beta) & ::= & \mathtt{F}\alpha, \beta \\
\mathrm{MR}_{\mathtt{F}}(\alpha, \beta) & ::= & \alpha, \mathtt{F}\beta \\
\mathrm{A}_{\mathtt{F}}(\alpha, \beta) & ::= & \mathtt{F}\alpha, \mathtt{F}\beta \\
\mathrm{UR}_{\mathtt{F}}(\alpha \to \alpha', \beta) & ::= & \mathtt{F}\alpha \to \alpha', \beta \\
\mathrm{UL}_{\mathtt{F}}(\alpha, \beta \to \beta') & ::= & \alpha, \mathtt{F}\beta \to \beta' \\
\mathrm{C}_{\mathtt{LR}}(\mathtt{L}\alpha, \mathtt{R}\beta) & ::= & (\alpha, \beta) \\
\mathrm{ER}_{\mathtt{R}}(\mathtt{R}(\alpha \to \alpha'), \beta) & ::= & \alpha \to \mathtt{R}\alpha', \beta \\
\mathrm{EL}_{\mathtt{R}}(\alpha, \mathtt{R}(\beta \to \beta')) & ::= & \alpha, \beta \to \mathtt{R}\beta'
\end{array}
$$

Figure 3 Possible type combinations in the form of a near CFG. Here, $a, b \in \star_0$, $\alpha, \beta, \tau \in \star$ and $\mathtt{F}, \mathtt{L}, \mathtt{R} \in \mathcal{F}(\mathcal{L})$ with $\mathtt{L} \dashv \mathtt{R}$.

$$
\begin{array}{c}
> = \lambda\varphi.\lambda x.\varphi x \\
< = \lambda x.\lambda\varphi.\varphi x \\
\mathrm{ML}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathtt{F}} \lambda a.M(a, y))x \\
\mathrm{MR}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathtt{F}} \lambda b.M(x, b))y \\
\mathrm{A}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathtt{F}} \lambda a.\lambda b.M(a, b))(x)\mathtt{<*>}y \\
\mathrm{UL}_{\mathtt{F}} = \lambda M.\lambda x.\lambda\varphi.M(x, \lambda b.\varphi(\eta_{\mathtt{F}} b))
\end{array}
\qquad
\begin{array}{c}
\mathrm{UR}_{\mathtt{F}} = \lambda M.\lambda\varphi.\lambda y.M(\lambda a.\varphi(\eta_{\mathtt{F}} a), y) \\
\mathrm{J}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.\mu_{\mathtt{F}} M(x, y) \\
\mathrm{C}_{\mathtt{LR}} = \\
\lambda M.\lambda x.\lambda y.\varepsilon_{\mathtt{LR}}(\mathtt{fmap}_{\mathtt{L}}(\lambda l.\mathtt{fmap}_{\mathtt{R}}(\lambda r.M(l, r))(y))(x)) \\
\mathrm{EL}_{\mathtt{R}} = \lambda M.\lambda\varphi.\lambda y.M(\Upsilon_{\mathtt{R}}\varphi, y) \\
\mathrm{ER}_{\mathtt{R}} = \lambda M.\lambda x.\lambda\varphi.M(x, \Upsilon_{\mathtt{R}}\varphi) \\
\mathrm{DN}_{\Downarrow} = \lambda M.\lambda x.\lambda y. \Downarrow M(x, y)
\end{array}
$$

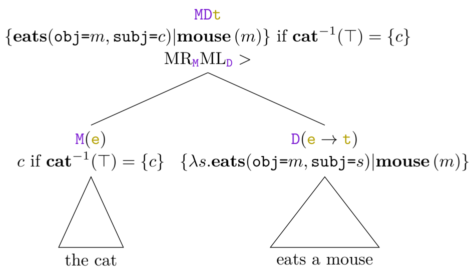
Figure 4 Denotations describing the effect of the combinators used in the grammar describing our combination modes presented in Figure 3

2. We introduce a structure for the semantic parse trees and their labels, based on the combination modes from [1].
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.

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 4.

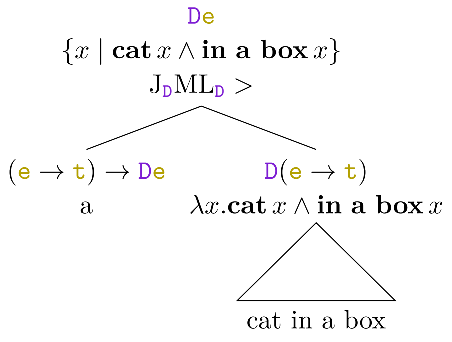
The main reason why denotations associated to combinators are needed, is to properly define how they actually do the combination of denotations. Those denotations are a direct translation of the judgements defining the notions of functors, applicatives, monads and thus are not specific to any denotation system, even though we use lambda-calculus to describe them. Some are duplicated for a left and right version to account for the fact CFGs are not actually symmetric in their "input" unlike intuitionistic inference rules.

This makes us able to compute the actual denotations associated to a sentence using our formalism, as presented in Figure 5. 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, they are actually modifying a combination mode via their denotation.



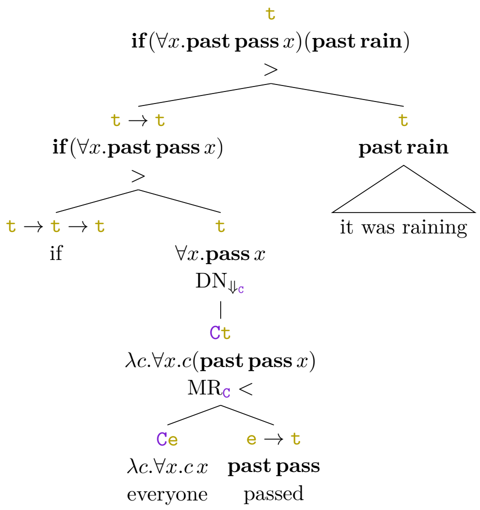
$\mathtt{MDt}$
$\{\mathbf{eats}(\mathtt{obj}{=}m, \mathtt{subj}{=}c) | \mathbf{mouse}\,(m)\}$ if $\mathbf{cat}^{-1}(\top) = \{c\}$
$\mathrm{MR}_{\mathtt{M}}\mathrm{ML}_{\mathtt{D}} >$
$\mathtt{M(e)}$
c if $\mathbf{cat}^{-1}(\top) = \{c\}$
$\mathtt{D(e \to t)}$
$\{\lambda s.\mathbf{eats}(\mathtt{obj}{=}m, \mathtt{subj}{=}s) | \mathbf{mouse}\,(m)\}$
the cat
eats a mouse

(a) Labelled tree representing the equivalent parsing diagram to 7



$\mathtt{De}$
$\{x \mid \mathbf{cat}\,x \wedge \mathbf{in\ a\ box}\,x\}$
$\mathrm{J}_{\mathtt{D}}\mathrm{ML}_{\mathtt{D}} >$
$\mathtt{(e \to t) \to De}$
a
$\mathtt{D(e \to t)}$
$\lambda x.\mathbf{cat}\,x \wedge \mathbf{in\ a\ box}\,x$
cat in a box

(b) Labelled tree representing the equivalent parsing diagram to 8



$\mathtt{t}$
$\mathbf{if}(\forall x.\mathbf{past\,pass}\,x)(\mathbf{past\,rain})$
$>$
$\mathtt{t \to t}$
$\mathbf{if}(\forall x.\mathbf{past\,pass}\,x)$
$>$
$\mathtt{t}$
$\mathbf{past\,rain}$
it was raining
$\mathtt{t \to t \to t}$
if
$\mathtt{t}$
$\forall x.\mathbf{pass}\,x$
$\mathrm{DN}_{\Downarrow\mathtt{C}}$
$\mathtt{Ct}$
$\lambda c.\forall x.c(\mathbf{past\,pass}\,x)$
$\mathrm{MR}_{\mathtt{C}} <$
$\mathtt{Ce}$
$\lambda c.\forall x.c\,x$
everyone
$\mathtt{e \to t}$
$\mathbf{past\,pass}$
passed

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

Figure 5 Examples of labelled parse trees for a few sentences.

This formalism gives us the following theorems:

▶ **Theorem 4.** *Parsing of a sentence with combination modes 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 $G_{\mathcal{T}}$. The syntactico-semantic system G constructed from the product of G_s and $G_{\mathcal{T}}$ has size $|G_s| \times |G_{\mathcal{T}}|$. Computing membership of a sentence to the language generated by G, is then in polynomial time if, and only if, finding membership to the language generated by G_s is done in polynomial time. Parsing the sentence is then done in polynomial time in the size of the input, $|G_s|$ and $|G_{\mathcal{T}}| = \mathcal{O}\left(|\mathcal{F}\left(\mathcal{L}\right)| + |\mathrm{Obj}\left(\mathcal{C}\right)|\right)$. ■

▶ **Theorem 5.** *Retrieving a pure denotation for a sentence is polynomial in the length of the sentence, given a polynomial time syntactic parsing structure and polynomial combinator denotations.*

To prove this theorem we need a short lemma on the size of the trees generated through our structure:

▶ **Lemma 6.** *Semantic parsing trees are quadratic in the length of the sentence.*

Proof. Let $m_{\mathcal{L}}$ be the maximum number of effects created by a word in $\mathcal{L}$. Since at any step i in the parsing, there can never be more than $m_{\mathcal{L}} \times i$ effects borne by the considered inputs, there is no need for more than $(2+c) \times m_{\mathcal{L}} \times (i+1)+1$ combinators where c is a constant dependent only on the language. Indeed, we will have at most one combinator among $\{\mathrm{ML}, \mathrm{MR}, \mathrm{A}, \mathrm{UR}, \mathrm{UL}\}$ per input effect, at most one of J and DN per output effects ($m_{\mathcal{L}} \times (i+1)$ at most), at most a fixed number c of modes between $\{\mathrm{C}, \mathrm{EL}, \mathrm{ER}\}$ which depends only on the number of adjunctions in the language. We get the wanted upper bound when adding the *base combinator*. Summing the steps for i, we get a quadratic upper bound on the number of combinators and thus on the tree size. ■

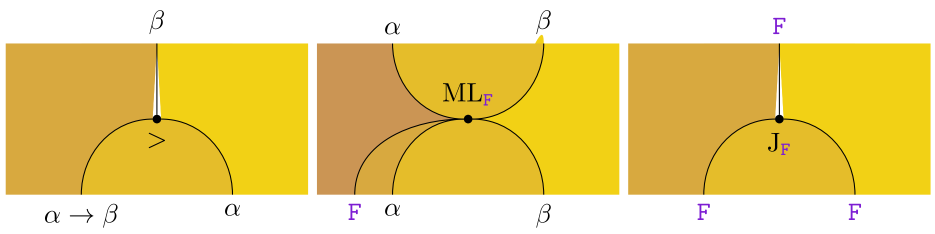


Figure 6 String Diagrammatic Representation of Combinator Modes $>$, ML and J

We can now return to the proof of the main result of this section:

Proof of Theorem 5. From Theorem 4 we can retrieve a semantic parse tree from a sentence in polynomial time in the input. Lemma 6 states that we have a polynomial number of combinator denotations to apply, all done in polynomial time by hypothesis. 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 sequencing of these steps yields a polynomial-time algorithm in the length of the input sentence. ◀

While we have gone the assumption that we have a CFG for our language, any type of polynomial-time structure could work, as long as it is at least as expressive as a CFG.

The *polynomial time combinators* assumption in Theorem 5 is not a complex assumption, this is for example true for denotations based on lambda-calculus, 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 the different `fmap` being in polynomial time for the same reason. This would also be true for denotations inspired by machine learning for example.

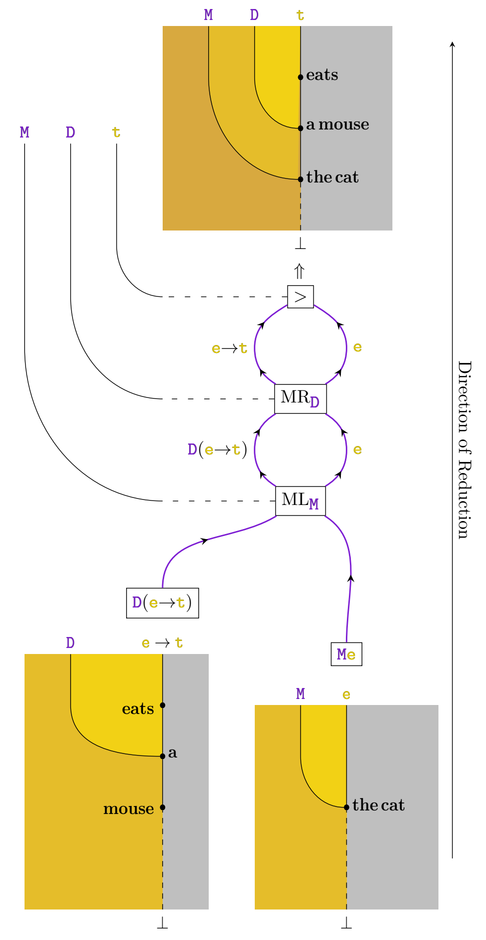


Figure 7 Representation of a parsing diagram for the sentence *the cat eats a mouse*. See Figure 5b for translation in a parse tree.

5.2 Diagrammatical Parsing

When considering [2] way of using string diagrams for syntactic parsing/reductions, we can see string diagrams as (yet) another way of writing our parsing rules. They are an expanded rewriting of labelled parsing trees[3] presented in [1], . In our typed category, we can see

[3] Point of view which connects this formalism nicely to the one of [9], preserving all their results inside our theory.

our combinators as natural transformations (2-cells): then we can see the different sets of combinators as different arity natural transformations. Combinators >, $\mathrm{ML}_{\mathtt{F}}$ and $\mathrm{J}_{\mathtt{F}}$ are represented in Figure 6. The coloring of the regions is purely for artistic rendition and will not be used for larger diagrams.

Understanding the diagrams could be thinking of them on an orthogonal plane to the ones of Section 4: we could use the syntactic version of the diagrams to model our parsing, according to the rules in Figure 3, and then combine the diagrams as shown in Figure 7, which highlights the *orthogonal* components. In this diagram we exactly see the sequence of combinations play out on the types of the words, and thus we also see what exact *stitch* would be needed to construct the effect diagram. Here we talk about *stitches* 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 computation, leading to the intervention of outside tools (combinators) which serve as *knitting needles*. To better understand what happens in those parsing diagrams, Figure 5 provides the translations in labelled trees of the parsing diagrams of Figures 7, 8 and 10.

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. Applying them to the actual *parsing* part of the diagram is done in the exact same way as in the CFG: we just add them where needed, and they will appear in the resulting denotation as a form of forced handling, in a sense, as shown in the result of Figure 8. It is interesting to note that the resulting diagram representing the sentence can visually be found in the connection strings that arise from the combinators.

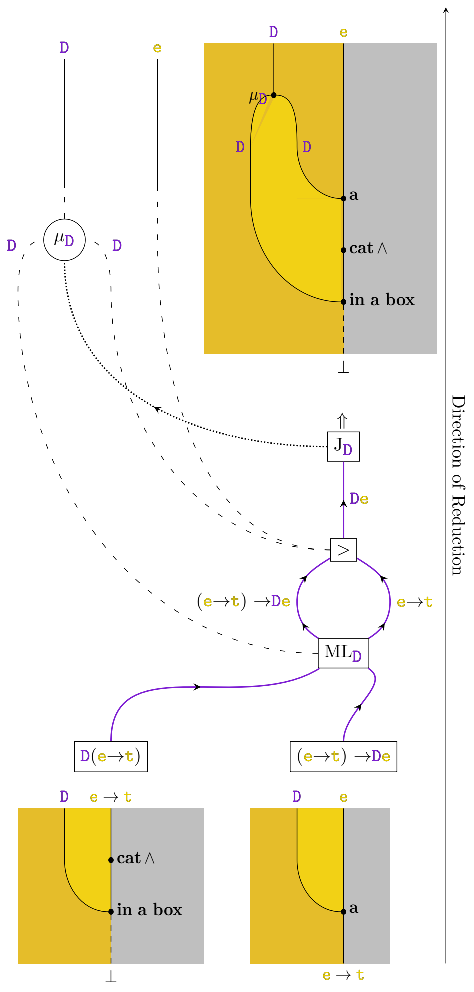


Figure 8 Example of a parsing diagram for the phrase *a cat in a box*, presenting the integration of unary combinators inside the connector line. See Figure 5b for translation in a parse tree.

Categorically, 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 [2] and [10], as it is seemingly a typed version, where we change the pregroup category for the typing category, taken with a product for representation of the English grammar representation, to accommodate for syntactic typing on top of semantic typing if it does not already encompass it. We have a first plane of string diagrams in the category $\mathcal{C}$ - our string diagrams for effect handling, as in Section 4 - and the second *orthogonal* plane of string diagrams on a larger category, with formal endofunctors labelled by the types in our typing category $\bar{\mathcal{C}}$ and formal natural transformations for the combinators defined in Figures 3 and 4. 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 endofunctors 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.

The main theoretical reason 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 stated in Theorem 3, string diagrams make it easy to compute normal forms when provided with a confluent reduction system. However, the just as useful graphical interpretation of string diagrams as easy to read expanded labelled parsing trees. Using orthogonal planes to visualise this interpretation cannot be well presented in a 3D space, and even less so on a page, so we suggest an interpretation based on actual strings: 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.

Figure 9 Example of a *Jacquard* knitwork. Photography and work courtesy of the author's mother.

You can create a new colour[4] thread from two (that's the base combinators). Creating a thicker one from two of the same colour is the result of the applicative mode and the monadic join. `fmap` puts aside a thread until a later step, the monadic unit adds a new thread to the pattern, and the co-unit and closure operators cut a thread which will no longer be used. Changing a thread by cutting it and making a knot at another point is what the eject combinators do.

This more tangible representation can be seen in Figure 10. The sections in the rectangle represent what happens when considering our combination step as implementing patterns inside a knitwork, as seen in Figure 9. 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.

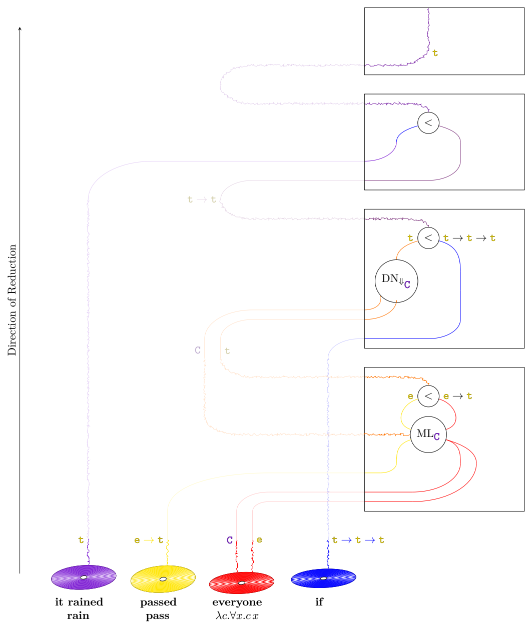


Figure 10 Knitting-like representation of the diagrammatic parsing of a sentence. See Figure 5c for the translation in a parse tree

[4] This is not how wool works, but one can also 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.

5.3 Rewriting Rules

In this section we study reductions for our diagrams that allows us to improve our time complexity by reducing the size of the grammar. This is done by looking at equations on sequences of combinators. In the worst case, there is no improvement in big o notation in the size of the sentence, but there is no loss.

Consider the case where we have the two arguments of our parsing step of type $\mathtt{F}\tau$ and $\mathtt{G}\tau'$. In that case we could either get a result with effects $\mathtt{FG}$ or with effects $\mathtt{GF}$. If those effects happen to be equal, which trivially will be the case when one of the effects is external (the plural or islands functors for example), the order of application does not matter and we choose to get the effect on the left side of the combinator first: $\mathrm{ML}_{\mathtt{F}}\mathrm{MR}_{\mathtt{G}}$ over $\mathrm{MR}_{\mathtt{F}}\mathrm{ML}_{\mathtt{F}}$.

There are sequence of modes that clearly encompass other ones the grammar notation for ease of explanation. One should not use the unit of a functor after using ML or MR, as that adds void semantics. Same things can be said for certain other derivations containing the lowering and co-unit combinators since they could in theory be applied at many points inside the derivation.

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:
  - $\mathrm{MR}_{\mathtt{F}}, \mathrm{A}_{\mathtt{F}}$
  - $\mathrm{A}_{\mathtt{F}}, \mathrm{ML}_{\mathtt{F}}$
  - $\mathrm{MR}_{\mathtt{F}}, j, \mathrm{ML}_{\mathtt{F}}$
  - $\mathrm{A}_{\mathtt{F}}, j, \mathrm{A}_{\mathtt{F}}$

▶ **Theorem 7.** *The rules proposed above yield equivalent results.*

Proof. 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 along Theorem 1: the only reason to keep two of the same effects and not join them is to at some point have something get in between the two. Joining and closure should then be done at earliest point in parsing where it can be done, and that is equivalent to later points because of Theorem 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 reductions described above amount to equational reductions for the string diagrams, as they are equivalent to specific sequences of 2-cells. This leads to the same algorithms developed in Section 4 being usable here: we just have a new improved version of Theorem 2: computing two different normal forms along the tensor product of our reduction schemes, which amounts to computing a larger normal form. Theorem 3 still stands with the improved system and thus, proving two parses are equal can be done in polynomial time. Moreover, considering the possible normal forms of syntactic reductions or denotational reductions adds ways to reduce our diagrams to normal forms.

6 Conclusion

The functional programming approach developed in [1] allows for increased expressiveness in the choice of denotations, especially from a purely theoretical point of view. In this paper we have successfully proven that such an approach is well-founded theoretically, but also that it doesn't come at the cost of comprehensibility or efficiency. Given the entirely theoretical denotations for common language objects (think of **cat** = **cat** as a definition), our methods might give enough latitude to semanticists to imagine more precise denotations without the cost of heavy statistical analyses, or at least, give tools to expand on them. Deriving our formalism from a theory necessitates only to understand what base combinators exist for the model: we build upon a basic semantic theory to increase its expressiveness.

Moreover, while our methods for implementing a type and effect system have been applied to natural language semantics, they could be applied in any language with purely compositional semantics. Of course, improvements can be made, in particular around the unorthodox use of effects to define what we have called higher-order constructs and scope islands, but also in integrating the theory in more complicated models of denotations, such as the ones learned through a neural network for example.

References

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

2 Bob Coecke, Mehrnoosh Sadrzadeh, and Stephen Clark. Mathematical Foundations for a Compositional Distributional Model of Meaning, March 2010. `arXiv:1003.4394`, `doi:10.48550/arXiv.1003.4394`.

3 Antonin Delpeuch and Jamie Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm, January 2022. `arXiv:1804.07832`, `doi:10.48550/arXiv.1804.07832`.

4 André Joyal and Ross Street. The geometry of tensor calculus, I. *Advances in Mathematics*, 88(1):55–112, July 1991. `doi:10.1016/0001-8708(91)90003-P`.

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

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

7 Paul-André Melliès and Noam 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, January 2015. ACM. `doi:10.1145/2676726.2676970`.

8 Paul-André Melliès and Noam Zeilberger. The categorical contours of the Chomsky-Sch\"utzenberger representation theorem. *Logical Methods in Computer Science*, Volume 21, Issue 2:13654, May 2025. `doi:10.46298/lmcs-21(2:12)2025`.

9 Isabella Senturia and Matilde Marcolli. The Algebraic Structure of Morphosyntax, June 2025.

10 Alexis Toumi and Giovanni de Felice. Higher-Order DisCoCat (Peirce-Lambek-Montague semantics), November 2023. `arXiv:2311.17813`, `doi:10.48550/arXiv.2311.17813`.

11 Philip 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, November 1989. Association for Computing Machinery. `doi:10.1145/99370.99404`.

12 Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the 2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA, September 2014. Association for Computing Machinery. `doi:10.1145/2633357.2633358`.

A Presenting a Language

In this appendix, we provide tables (11a and 11b) describing the modeling of a subset of the English language in our formalism.

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	$\mathtt{e}$	$\mathbf{j} \in \mathrm{Var}$
sleep	$\mathtt{e} \to \mathtt{t}$	$\lambda x.\mathbf{sleep}\, x$
chase	$\mathtt{e} \to \mathtt{e} \to \mathtt{t}$	$\lambda o.\lambda s.\mathbf{chase}\,(o)\,(s)$
be	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$	$\lambda p.\lambda x.px$
it	$\mathtt{Ge}$	$\lambda g.g_0$
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	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Ce}$	$\lambda p.\lambda c.\neg\exists x.px \wedge c\, x$
$\cdot$, **a** $\cdot$	$\mathtt{e} \to (\mathtt{e} \to \mathtt{t}) \to \mathtt{We}$	$\lambda x.\lambda p.\, \langle x, px\rangle$

(a) Lexicon for a subset of the English language

Constructor	`fmap`	Interpretation
$\mathtt{G}(\tau) = \mathtt{r} \to \tau$	$\mathtt{G}\varphi\,(x) = \lambda r.\varphi\,(xr)$	Read
$\mathtt{W}(\tau) = \tau \times \mathtt{t}$	$\mathtt{W}\varphi\,(\langle a, p\rangle) = \langle \varphi a, p\rangle$	Write
$\mathtt{S}(\tau) = \{\tau\}$	$\mathtt{S}\varphi\,(\{x\}) = \{\varphi(x)\}$	Powerset
$\mathtt{C}(\tau) = (\tau \to \mathtt{t}) \to \mathtt{t}$	$\mathtt{C}\varphi\,(x) = \lambda c.x\,(\lambda a.c\,(\varphi a))$	Continuation
$\mathtt{D}(\tau) = \mathtt{s} \to \mathtt{S}(\tau \times \mathtt{s})$	$\mathtt{D}\varphi\,(\lambda s.\, \{\langle x, x + s\rangle \mid px\}) = \lambda s.\, \{\langle \varphi x, \varphi x + s\rangle \mid px\}$	State
$\mathtt{M}(\tau) = \tau + \bot$	$\mathtt{M}\varphi\,(x) = \begin{cases} \varphi\,(x) & \text{if } \Gamma \vdash x : \tau \\ \# & \text{if } \Gamma \vdash x : \# \end{cases}$	Maybe

(b) Definition of a few functors, with their map on functions

Figure 11 Presentation of a lambda-calculus lexicon for the English language