

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 expressivity 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 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 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 effi-



© Anonymous author(s);

licensed under Creative Commons License CC-BY 4.0

42nd Conference on Very Important Topics (CVIT 2016).

Editors: John Q. Open and Joan R. Access; Article No. 23; pp. 23:1–23:16

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

cient 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 $\text{Obj}(\bar{\mathcal{C}})$, 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 $\star = \text{Obj}(\bar{\mathcal{C}}) / \sim$ where \sim 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. $\text{Obj}(\mathcal{C})$. In contexts of polymorphism, we identify \star_0 to the adequate subset of \star . In this paradigm, constant objects (or results of fully handled computations) are functions with type $\perp \rightarrow \tau$ 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-Monads².
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 [12]. 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 use of handlers as annotations to the semantic tree of the sentence is an appropriate formalism. As considered by [13], handlers are to be seen as natural transformations describing the free monad on an algebraic effect. Considering handlers as so, allows us to directly handle our computations inside our typing system, by “transporting” our functors one order higher up without loss of information or generality since all our functors undergo the same transformation. Using the framework proposed in [11] we simply need to create handlers for our effects/functors and we will then have in our language the result needed. The only thing we will require from an algebraic handler h is that for any applicative functor of unit η , $h \circ \eta = \text{id}$.

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 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, by simply saying that those constructs are predicate modifiers passed down (with or without side effects) to the arguments of predicates:

$$\begin{aligned} \text{future}(\text{be})(\text{arg}_1, \text{arg}_2) &\xrightarrow{\eta} \text{future}(\text{be})(\text{arg}_2)(\text{future}(\text{arg}_1)) \\ &\xrightarrow{\eta} \text{future}(\text{be})(\text{future}(\text{arg}_2))(\text{future}(\text{arg}_1)) \end{aligned}$$

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.

3.1.3.3 Monad Transformers

In [1], 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

$$\begin{array}{c}
\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash F : S \subseteq \star \quad \overbrace{\tau \in S}^{\exists \tau' \in S, \tau \preceq \tau'}}{\Gamma \vdash Fx : F\tau \preceq \tau} \text{Cons} \\
\\
\frac{\Gamma \vdash x : \tau \quad \tau \in \star_0}{\Gamma \vdash Fx : F\tau \notin \star_0} FT_0 \\
\\
\frac{\Gamma \vdash x : F\tau_1 \quad \Gamma \vdash \varphi : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \varphi x : F\tau_2} \text{fmap} \\
\\
\frac{\Gamma \vdash x : A\tau_1 \quad \Gamma \vdash \varphi : A(\tau_1 \rightarrow \tau_2)}{\Gamma \vdash \varphi x : A\tau_2} \langle * \rangle \\
\\
\forall F \xRightarrow{\theta} G, \frac{\Gamma \vdash x : F\tau \quad \Gamma \vdash G : S' \subseteq \star \quad \tau \in S'}{\Gamma \vdash x : G\tau} \text{nat}
\end{array}$$

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

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.

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 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.

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

In the diagram of Figure 2, each string corresponds to a functorial effect or type layer applied during parsing. The base type string \mathfrak{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 \mathfrak{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 artistic touch.

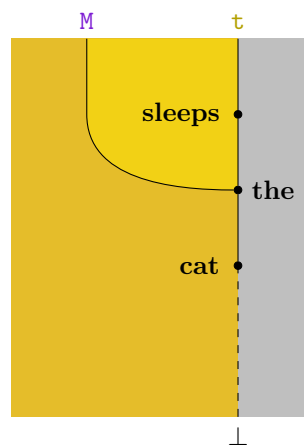


Figure 2 String diagram for the sentence *the cat sleeps*.

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 $nat = (nat.h, nat.in, nat.out)$. This gives a naive 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 datastructure supports the linear and polynomial time algorithms proposed with the structure by [3]. In particular our structure can be normalized in time $\mathcal{O}(n \times \sup_i |D.L(i).in| + |D.L(i).out|)$, which depends on our lexicon but most of the times will be linear time.

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:

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

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

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

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 $\mu(\mu(TT), T) \rightarrow \mu(T\mu(TT))$ so that the algorithm terminates. We use the same convention for the comonadic equations. The validity conditions are as easy to define for the monadic equations as for the *snake* equations when considering diagrams in *right* normal forms. Then, for 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 non-determinism as possible, and reducing time complexity of our parsing strategies.

5.1 Syntactic-Semantic Parsing

Using a naïve strategy on the trees yields an exponential algorithm, the best way to improve the algorithm, is to directly leave the naïve strategy. To do so, we could simply extend the grammar system used to do the syntactic part of the parsing. In this section, we will take the example of a CFG since it suffices to create our typing combinators, 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:

Here, $a, b \in \star_0$, $\alpha, \beta, \tau \in \star$ and $\mathbf{F}, \mathbf{L}, \mathbf{R} \in \mathcal{F}(\mathcal{L})$ with $\mathbf{L} \dashv \mathbf{R}$.

| | | | |
|--|--|---|---|
| $>, b$ | $::= (a \rightarrow b), a$ | $\text{ML}_{\mathbf{F}}(\alpha, \beta)$ | $::= \mathbf{F}\alpha, \beta$ |
| $<, b$ | $::= a, (a \rightarrow b)$ | $\text{MR}_{\mathbf{F}}(\alpha, \beta)$ | $::= \alpha, \mathbf{F}\beta$ |
| $\wedge, a \rightarrow \mathbf{t}$ | $::= (a \rightarrow \mathbf{t}), (a \rightarrow \mathbf{t})$ | $\text{A}_{\mathbf{F}}(\alpha, \beta)$ | $::= \mathbf{F}\alpha, \mathbf{F}\beta$ |
| $\vee, a \rightarrow \mathbf{t}$ | $::= (a \rightarrow \mathbf{t}), (a \rightarrow \mathbf{t})$ | $\text{UR}_{\mathbf{F}}(\alpha \rightarrow \alpha', \beta)$ | $::= \mathbf{F}\alpha \rightarrow \alpha', \beta$ |
| | | $\text{UL}_{\mathbf{F}}(\alpha, \beta \rightarrow \beta')$ | $::= \alpha, \mathbf{F}\beta \rightarrow \beta'$ |
| $\mathbf{J}_{\mathbf{F}} \mathbf{F}\tau$ | $::= \mathbf{F}\mathbf{F}\tau$ | $\text{C}_{\mathbf{LR}}(\mathbf{L}\alpha, \mathbf{R}\beta)$ | $::= (\alpha, \beta)$ |
| $\text{DN}_{\mathbf{C}} \tau$ | $::= \mathbf{C}_{\tau} \tau$ | $\text{ER}_{\mathbf{R}}(\mathbf{R}(\alpha \rightarrow \alpha'), \beta)$ | $::= \alpha \rightarrow \mathbf{R}\alpha', \beta$ |
| | | $\text{EL}_{\mathbf{R}}(\alpha, \mathbf{R}(\beta \rightarrow \beta'))$ | $::= \alpha, \beta \rightarrow \mathbf{R}\beta'$ |

■ **Figure 3** Possible type combinations in the form of a near CFG

| | |
|---|--|
| $\geq = \lambda\varphi.\lambda x.\varphi x$ | $\text{UR}_{\mathbf{F}} = \lambda M.\lambda\varphi.\lambda y.M(\lambda a.\varphi(\eta_{\mathbf{F}}a), y)$ |
| $\leq = \lambda x.\lambda\varphi.\varphi x$ | $\mathbf{J}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.\mu_{\mathbf{F}}M(x, y)$ |
| $\text{ML}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.(\mathbf{fmap}_{\mathbf{F}}\lambda a.M(a, y))x$ | $\text{C}_{\mathbf{LR}} =$ |
| $\text{MR}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.(\mathbf{fmap}_{\mathbf{F}}\lambda b.M(x, b))y$ | $\lambda M.\lambda x.\lambda y.\varepsilon_{\mathbf{LR}}(\mathbf{fmap}_{\mathbf{L}}(\lambda l.\mathbf{fmap}_{\mathbf{R}}(\lambda r.M(l, r))(y))(x))$ |
| $\text{A}_{\mathbf{F}} = \lambda M.\lambda x.\lambda y.(\mathbf{fmap}_{\mathbf{F}}\lambda a.\lambda b.M(a, b))(x) \leq * > y$ | $\text{EL}_{\mathbf{R}} = \lambda M.\lambda\varphi.\lambda y.M(\Upsilon_{\mathbf{R}}\varphi, y)$ |
| $\text{UL}_{\mathbf{F}} = \lambda M.\lambda x.\lambda\varphi.M(x, \lambda b.\varphi(\eta_{\mathbf{F}}b))$ | $\text{ER}_{\mathbf{R}} = \lambda M.\lambda x.\lambda\varphi.M(x, \Upsilon_{\mathbf{R}}\varphi)$ |
| | $\text{DN}_{\Downarrow} = \lambda M.\lambda x.\lambda y.\Downarrow M(x, y)$ |

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

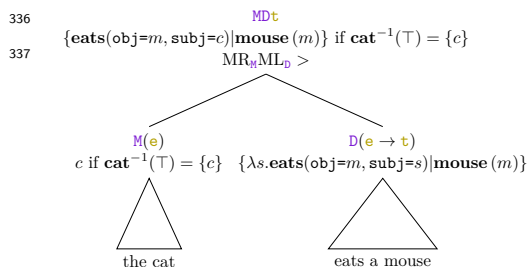
1. We reintroduce the grammar defining the type and effect system.
2. We introduce a structure for the semantic parse trees and their labels, the combination modes from [1].
3. We introduce rules for basic type combinations.
4. We introduce rules for higher-order unary type combinators.
5. We introduce rules for higher-order binary type combinators.

We do not prove here that these typing rules cannot be written in a simpler syntactic structure, but it is easy to see why a regular expression would not work, and since we need trees, to express our full system, the best we can do would be to disambiguate the grammar.

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 we need to get denotations associated to combinators, is to properly define how they actually do the combination of denotations. The important thing on those derivation 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 5. Note that



(a) Labelled tree representing the equivalent parsing diagram to 6

the order of combination modes is not actually the same as the one that would come from the grammar. The reason why will become more apparent when string diagrams for parsing are introduced in the next section, but simply, this comes from the fact that while we think of ML and MR as reducing the number of effects on each side (and this is the correct way to think about those), this is not actually how its denotation works, but there is no real issue with that. This formalism gives us the following theorems:

► **Theorem 4.** *Semantic parsing of a sentence is polynomial in the length of the sentence and the size of the type system and syntax system.*

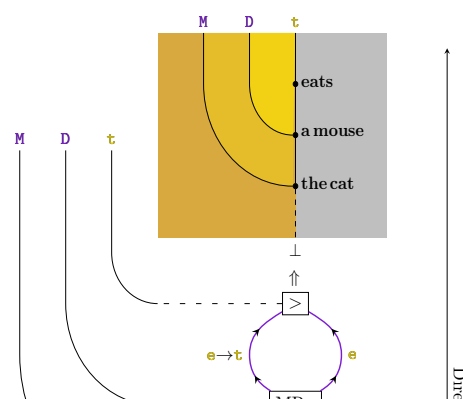
Proof. Suppose we are given a syntactic generating structure G_s along with our type combination grammar G_τ . The system G that can be constructed from the (tensor) product of G_s and G_τ has size $|G_s| \times |G_\tau|$. Indeed, we think of its rules as a rule for the syntactic categories and a rule 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 $|G_\tau|$. ■

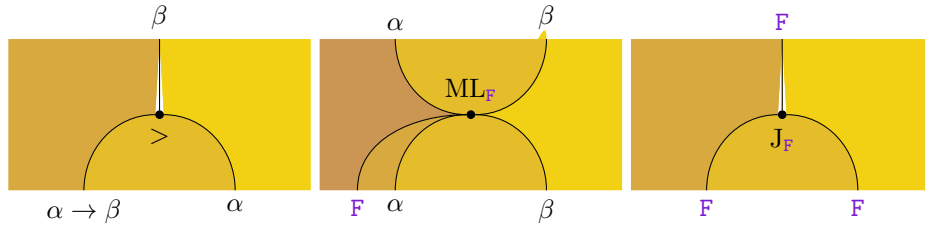
► **Theorem 5.** *Retrieving a pure denotation for a sentence can be done in polynomial time in the length of the sentence, given a polynomial time syntactic parsing algorithm and polynomial time combinators.*

Proof. We have proved in Theorem 4 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 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. ■

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 more expressive than a CFG.





■ **Figure 7** String Diagrammatic Representation of Combinator Modes $>$, ML and J

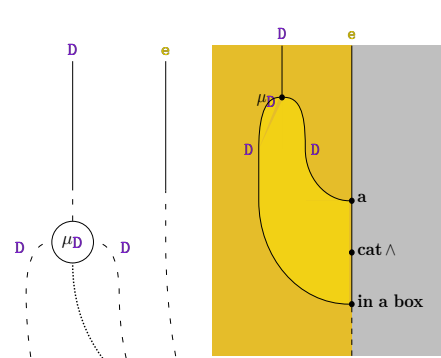
384 The *polynomial time combinators* assumption
 385 in the following theorem is not a complex as-
 386 sumption, this is for example true for denota-
 387 tions based on lambda-calculus, with function
 388 application being linear in the number of uses of
 389 variables in the function term, which in turn is
 390 linear in the number of terms used to construct
 391 the function term and thus of words, and the dif-
 392 ferent `fmap` being in constant time for the same
 393 reason.

394 5.2 Diagrammatic Parsing

395 When considering [2] way of using string dia-
 396 grams for syntactic parsing/reductions, we can
 397 see them as (yet) another way of writing our pars-
 398 ing rules. In our typed category, we can make
 399 see our combinators as natural transformations
 400 (2-cells): then we can see the different sets of
 401 combinators as different arity natural transforma-
 402 tions. $>$, ML_F and J_F are represented in Figure
 403 7, up to the coloring of the regions, because that
 404 is purely for an artistic rendition.

405 Now, a way to see this would be to think of
 406 this as an orthogonal set of diagrams to the ones
 407 of Section 4: we could use the syntactic version
 408 of the diagrams to model our parsing, according
 409 to the rules in Figure 3, and then combine the
 410 diagrams as shown in Figure 6, which highlights
 411 why we can consider the diagrams orthogonal. In this figure we exactly see the sequence of
 412 reductions play out on the types of the words, and thus we also see what exact *quasi-braiding*
 413 would be needed to construct the effect diagram. Here we talk about *quasi-braiding* because,
 414 in a sense, we use 2-cells to do braiding-like operations on the strings, and don't actually
 415 allow for braiding inside the diagrammatic computation. To better understand what happens
 416 in those parsing diagrams, Figure 5 provides the translations in labelled trees of the parsing
 417 diagrams of Figures 6, 8 and 10.

418 Categorically, what this means is that we
 419 start from a meaning category \mathcal{C} , our typing cat-
 420 egory, and take it as our grammatical category.



421 This is a form of extension on the monoidal ver-
 422 sion by [2], as it is seemingly a typed version,
 423 where we change the pregroup category for the
 424 typing category, possibly taken with a product
 425 for representation of the English grammar rep-
 426 resentation, to accommodate for syntactic typing
 427 on top of semantic typing. This is, again, just
 428 another rewriting of our typing rules.

429 We have a first axis of string diagrams in
 430 the category \mathcal{C} - our string diagrams for effect
 431 handling, as in Section 4 - and a second *ortho-*
 432 *gonal* axis of string diagrams on a larger category,
 433 with endofunctors labelled by the types in our
 434 typing category $\bar{\mathcal{C}}$ and with natural transform-
 435 ations mapping the combinators defined in Fig-
 436 ures 3 and 4. The category in which we consider
 437 the second-axis string diagrams does not have a
 438 meaning in our compositional semantics theory,
 439 and to be more precise, we should talk about
 440 1-cells and 2-cells instead of functors and natural
 441 transformations, to keep in the idea that this is
 442 really just a diagrammatic way of computing and
 443 presenting the operations that are put to work
 444 during semantic parsing: A good approach to
 445 these diagrams is that they are an extension of
 446 the labelled parse trees presented in [1], forming
 447 a full diagrammatic calculus system for semantic
 448 parsing.

449 For the combinators J, DN and C, which are
 450 applied to reduce the number of effects inside
 451 a denotation, it might seem less obvious how
 452 to include them. Applying them to the actual
 453 *parsing* part of the diagram is done in the exact
 454 same way as in the CFG: we just add them where needed, and they will appear in the
 455 resulting denotation as a form of forced handling, in a sense, as shown in the result of Figure
 456 8. It is interesting to note that the resulting diagram representing the sentence can visually
 457 be found in the connection strings that arise from the combinators.

458 The main reason why this point of view of diagrammatic parsing is useful will be clear
 459 when looking at the rewriting rules and the normal forms they induce, because, as seen before,
 460 string diagrams make it easy to compute normal forms, when provided with a confluent
 461 reduction system.

462 The other reason being the tangible interpretation of how things work underlying the
 463 idea of a string diagram:

464 Suppose you're knitting a rainbow scarf. You
 465 have multiple threads (the different words) of the
 466 different colours (their types and effects) you're
 467 using to knit the scarf. When you decide to
 468 change the color you take the different threads



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

you have been using, and mix them up. You can create a new colour³ 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 10.

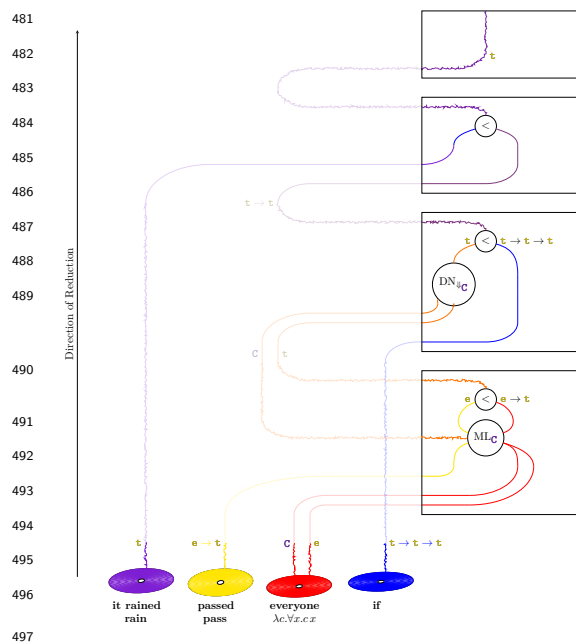


Figure 10 Knitting-like representation of the diagrammatic parsing of a sentence. See Figure 5c for the translation in a parse tree

(or possibly more). 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), 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.

Then 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:

We use DN when we have not used any of the following, in all derivations:

The sections in the rectangle represent what happen 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.

5.3 Rewriting Rules

Here we provide a rewriting system on that allows us to improve our time complexity by reducing the size of the 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 consider the case where we have presuppositions (or sub-trees) for the two arguments of our parsing step, that are of type $F\tau$ and $G\tau'$. In that case we could either get a result with effects FG or with effects GF

³ This is not how wool works, but if 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.

- 511 ■ m_F, DN, m_F where $m \in \{MR, ML\}$ 514 ■ ML_F, DN, A_F
 512 ■ ML_F, DN, MR_F
 513 ■ A_F, DN, MR_F 515 ■ C

516 We use J if we have not used any of the following, for $j \in \{\varepsilon, J_F\}$

- 517 ■ $\{m_F, j, m_F\}$ where $m \in \{MR, ML\}$ 522 ■ If F is commutative as a monad:
 518 ■ ML_F, j, MR_F 523 ■ MR_F, A_F
 519 ■ A_F, j, MR_F , 524 ■ A_F, ML_F
 520 ■ ML_F, j, A_F 525 ■ MR_F, j, ML_F
 521 ■ k, C for $k \in \{\varepsilon, A_F\}$ 526 ■ A_F, j, A_F

527 ► **Theorem 6.** *The rules proposed above yield equivalent results.*

528 **Proof.** For the first point, it is obvious since based on an equality. For the second point:
 529 The rules about not using combinators UL and UR come from the notion of handling and
 530 granting termination and decidability to our system. The rules about adding J and DN after
 531 moving two of the same effect from the same side (i.e. $MLML$ or $MRMR$) are normalization
 532 of Theorem 1. Indeed, in the denotation, the only reason to keep two of the same effects and
 533 not join them is to at some point have something get in between the two. Joining and closure
 534 should then be done at earliest point in parsing where it can be done, and that is equivalent
 535 to later points because of the elevator equations, or Theorem 1. The last set of rules follows
 536 from the following: we should not use $JMLMR$ instead of A , as those are equivalent because
 537 of the equation defining them. The same thing goes for the other two, as we should use the
 538 units of monads over applicative rules and $fmap$. ■

539 When using our diagrammatic approach to parsing, we can write all the reductions
 540 described above to our paradigm: it simply amounts to constructing a set of equational
 541 reductions for the string diagrams. This leads to the same algorithms developed in Section
 542 4 being usable here: we just have a new improved version of Theorem 2 which adds the
 543 normal forms specified in this section to the newly added *orthogonal* axis of diagrammatic
 544 computations. What we're actually doing is computing two different normal forms along
 545 the tensor product of our reduction schemes, which amounts to computing a larger normal
 546 form. Theorem 3 still stands with the improved system and thus, proving two parses are
 547 equal can be done in polynomial time. Moreover, considering the possible normal forms of
 548 syntactic reductions or denotational reductions adds ways to reduce our diagrams to normal
 549 forms. Of course, this does not mean that our system is complete, as there might still be
 550 many possible reductions to be found, although this would not reduce the complexity of the
 551 algorithm and is more of a thought experiment.

552 6 Conclusion

553 The functional programming approach developed in [1] allows for increased expressiveness in
 554 the choice of denotations, especially from a purely theoretical point of view. In this paper
 555 we have successfully prove that it is well-founded theoretically, but also that it doesn't come
 556 at the cost of comprehensibility or efficiency.

557 Moreover, while our methods for implementing a type and effect system have been applied
 558 to natural language semantics, it could be applied in any language with purely compositional
 559 semantics. Of course, there are still improvements that can be made, in particular around
 560 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. In that case, it would suffice to understand what base combinators exist for the model to implement the formalism we described.

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-Schützenberger 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 Birthe van den Berg and Tom Schrijvers. A framework for higher-order effects & handlers. *Science of Computer Programming*, 234:103086, May 2024. doi:10.1016/j.scico.2024.103086.
- 12 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.
- 13 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 | $e \rightarrow t$ | $\lambda x.\mathbf{planet} \ x$ |
| Generalizes to common nouns | | |
| carnivorous | $(e \rightarrow t)$ | $\lambda x.\mathbf{carnivorous} \ x$ |
| Generalizes to predicative adjectives | | |
| skillful | $(e \rightarrow t) \rightarrow (e \rightarrow t)$ | $\lambda p.\lambda x.px \wedge \mathbf{skillful} \ x$ |
| Generalizes to predicate modifier adjectives | | |
| Jupiter | e | $\mathbf{j} \in \text{Var}$ |
| sleep | $e \rightarrow t$ | $\lambda x.\mathbf{sleep} \ x$ |
| chase | $e \rightarrow e \rightarrow t$ | $\lambda o.\lambda s.\mathbf{chase} \ (o) \ (s)$ |
| be | $(e \rightarrow t) \rightarrow e \rightarrow t$ | $\lambda p.\lambda x.px$ |
| it | $\mathbf{G}e$ | $\lambda g.g_0$ |
| the | $(e \rightarrow t) \rightarrow \mathbf{M}e$ | $\lambda p.x \text{ if } p^{-1}(\top) = \{x\} \text{ else } \#$ |
| a | $(e \rightarrow t) \rightarrow \mathbf{D}e$ | $\lambda p.\lambda s.\{ \langle x, x \# s \rangle \mid px \}$ |
| no | $(e \rightarrow t) \rightarrow \mathbf{C}e$ | $\lambda p.\lambda c.\neg \exists x.px \wedge c \ x$ |
| $\cdot, \mathbf{a} \cdot$ | $e \rightarrow (e \rightarrow t) \rightarrow \mathbf{W}e$ | $\lambda x.\lambda p.\langle x, px \rangle$ |

(a) Lexicon for a subset of the English language

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

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

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