# 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-


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. We will go back later on the differences between their Lambek inspired grammars and our more abstract way of looking at the semantic parsing of a sentence.

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.

Let $\mathcal{O}(\mathcal{L})$ be the set of words in the language whose semantic representation is a low-order function and $\mathcal{F}(\mathcal{L})$ the set of words whose semantic representation is a functor or high-order function. Our goals here are to describe more formally, using a categorical vocabulary, the environment in which the typing system for our language will exist, and how we connect words and other linguistic objects to the categorical formulation.

### 3.1 Typing Category

#### 3.1.1 Types

Let $\mathcal{C}$ be a closed cartesian category, which is used to represent the domain of types for our domain of uneffectful denotations. This represents our main typing system, consisting of types for words $\mathcal{O}(\mathcal{L})$ that can be expressed without effects (see Figure 10a for an example). Remember that $\mathcal{C}$ contains a terminal object $\perp$ representing 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 in a finite number from a finite number of functorial applications from an object of $\mathcal{C}$.

Formally, we consider for our types the quotient set $\star = \mathrm{Obj}\left(\bar{\mathcal{C}}\right)/\mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ does not induce an equivalence relation on $\mathrm{Obj}\left(\bar{\mathcal{C}}\right)$ but a preorder, we consider the chains obtained by the closure of the relation $x \succeq y \Leftrightarrow \exists F, y = F(x)$ (which should be seen as a subtyping relation as proposed in [7]). We also define $\star_0$ to be the set obtained when considering types which have not yet been *affected*, that is $\mathrm{Obj}(\mathcal{C})$. In contexts of polymorphism, we identify $\star_0$ to the adequate subset of $\star$. In this paradigm, constant objects (or results of fully done

computations) are functions with type $\bot \to \tau$ which we will denote directly by $\tau \in \star_0$. 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 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 are provided in Figure 10b.

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 conversion* judgement:

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

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

### 3.1.3 Natural Transformations

We could add judgements directly for adjunctions, but we instead add judgements for natural transformations, as adjunctions are natural transformations which arise from *natural* settings. While in general we do not want to find natural transformations, we want to be able to express these in three situations:

1. Adjunctions.
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.

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

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 use of handlers as annotations to the semantic tree of the sentence is an appropriate formalism. As considered by [12], 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 [10] 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 $\eta$, $h \circ \eta = \mathrm{id}$.

Note that the choice of the handler being part of the lexicon or the parser over the other is a philosophical question more than a semantical one, as both options will result in semantically equivalent models, the only difference will be in the way we consider the resolution of effects. This 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 $\Pi$ 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:

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

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 4c 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 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.

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

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

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

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

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

**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-derzive the subtyping judgement to allow for its implementation. Note that here, the syntax is not taken into account: a function is always written left of its arguments, whether or not they are actually in that order in the sentence.

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.

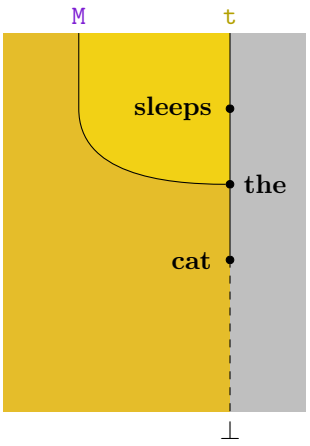
## 4.1 String Diagram Modelisation of Sentences

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 $\mathbb{1} \to \mathcal{C}$ is equivalent to choosing an object in $\mathcal{C}$, and a natural transformation between two such functors $\tau_1, \tau_2$ 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.




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]). *A well-formed equation between morphism terms in the language of monoidal categories follows from the axioms of monoidal categories if and only if it holds, up to planar isotopy, in the graphical language.*

Secondly, let us now look at a few of the equations that arise from the commutation of certain class of diagrams:

**The *Snake* Equations** are a rewriting of the categorical diagrams which are the defining properties of an adjunction.

**The *(co-)Monadic* Equations** are the string diagrammatic translation of the properties of unitality and associativity of the monad. Similarly, there are co-monadic equations which are the categorical dual of the previous equations.

This set of equations, when added to our reduction rules from the Section 5 explain all the different reductions that can be made to limit non-determinism in our parsing strategies. Indeed, considering the equivalence relation $\mathcal{R}$ freely generated from the equations defined above and the equivalence relationship $\mathcal{R}'$ of planar isotopy from Theorem 1, we get a set of normal forms $\mathcal{N}$ from the set of all well-formed parsing diagrams $\mathcal{D}$ defined by: $\mathcal{N} = (\mathcal{D}/\mathcal{R})\,/\mathcal{R}'$

## 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.

[3] proposed a combinatorial description to check in linear time for equality under Theorem 1. However, this model does not suffice to account for all of our equations, especially as labelling will influence the equations for monads, comonads and adjunctions. To provide with more flexibility (in exchange for a bit of time complexity) we use the description provided and change the description of inputs and outputs of each 2-cell by adding types and enforcing types. In this section we formally describe the data structure we propose, as well as algorithms for validity of diagrams and a system of rewriting that allows us to compute the normal forms for our system of effects.

### 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) along with the number of input strings, for each 2-cell the following information:

- Its horizontal position: the number of strings that are right of it.
- 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 $nat = (nat.\mathrm{h}, nat.\mathrm{in}, nat.\mathrm{out})$. From this, we can derive a naive algorithm to check if a string diagram is valid or not.

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

### 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. This also means that trivially our reduction system computes normal forms: it suffices to re-apply the algorithm for recumbent equivalence after the rest of equational reduction is done. Moreover, note that all our reductions are either incompatible or commutative, which leads to a confluent reduction system, and the well definition of our normal forms:

▶ **Theorem 2** (Confluence). *Our reduction system is confluent and therefore defines normal forms:*

1. *Right reductions are confluent and therefore define* right *normal forms for diagrams under the equivalence relation induced by exchange.*
2. *Equational reductions are confluent and therefore define* equational *normal forms for diagrams under the equivalence relation induced by exchange.*

Before proving the theorem, let us first provide the reductions for the different equations for our description of string diagrams.

**The Snake Equations** First, let's see when we can apply the equation for $\mathrm{id}_L$ 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:

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

This comes from the fact that we can't send either $\varepsilon$ above $\eta$ using right reductions and that there cannot be any natural transformations between the two. Obviously the equation for $\mathrm{id}_R$ 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 we need to *right* normalize before and after this method, and that this is done in linear time, our theorem is proved. ■

## 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

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

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

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

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

**Figure 2** Possible type combinations in the form of a near CFG

$$
\begin{array}{c}
>= \lambda\varphi.\lambda x.\varphi x \\
<= \lambda x.\lambda\varphi.\varphi x \\
\mathrm{ML}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathtt{F}}\lambda a.M(a, y))x \\
\mathrm{MR}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathtt{F}}\lambda b.M(x, b))y \\
\mathrm{A}_{\mathtt{F}} = \lambda M.\lambda x.\lambda y.(\mathtt{fmap}_{\mathtt{F}}\lambda a.\lambda b.M(a, b))(x)\mathtt{<*>}y \\
\mathrm{UL}_{\mathtt{F}} = \lambda M.\lambda x.\lambda\varphi.M(x, \lambda b.\varphi(\eta_{\mathtt{F}} b))
\end{array}
$$

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

**Figure 3** Denotations describing the effect of the combinators used in the grammar describing our combination modes presented in Figure 2

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 2, 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:

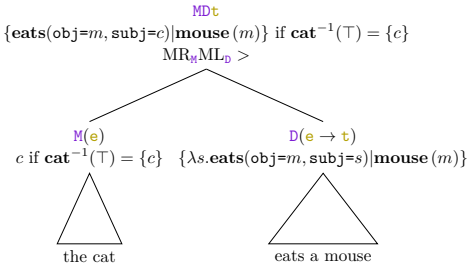
1. We reintroduce the grammar defining the type and effect system.
2. We introduce a structure for the semantic parse trees and their labels, the combination modes from [1].
3. We introduce rules for basic type combinations.
4. We introduce rules for higher-order unary type combinators.
5. We introduce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

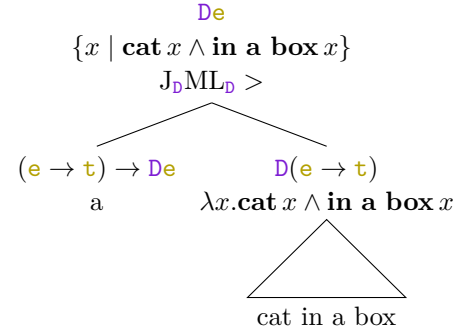
Each of these combinators can be, up to order, associated with a inference rule, and, as such, with a higher-order denotation, which explains the actual effect of the combinator, and are described in Figure 3.

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,

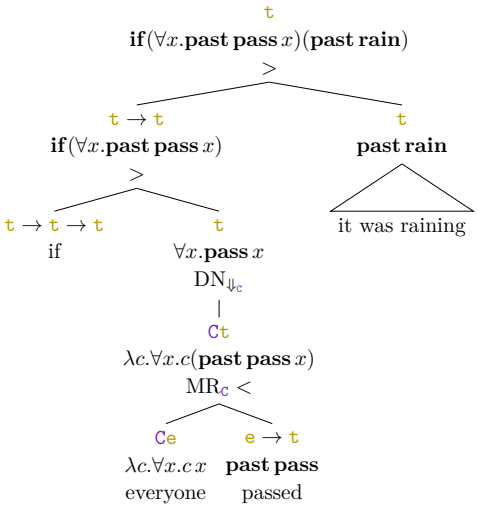
applicatives, monads and thus are not specific to any denotation system, even though we will use lambda-calculus styled denotations to describe them.




**(a)** Labelled tree representing the equivalent parsing diagram to 5




**(b)** Labelled tree representing the equivalent parsing diagram to 7




**(c)** Labelled tree representing the equivalent parsing diagram to 9

**Figure 4** Examples of labeled parse trees for a few sentences.

This makes us able to compute the actual denotations associated to a sentence using our formalism, as presented in figure 4. Note that the order of combination modes is not actually the same as the one that would come from the grammar. The reason why will become more apparent when string diagrams for parsing are introduced in the next section, but simply, this comes from the fact that while we think of ML and MR as reducing the number of effects on each side (and this is the correct way to think about those), this is not actually how its denotation works, but there is no real issue with that. 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 $G_\tau$. The system $G$ that can be constructed from the (tensor) product of $G_s$ and $G_\tau$ 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. ■

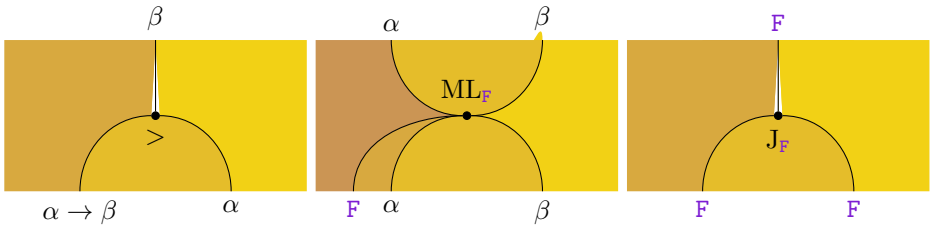



**Figure 6** String Diagrammatic Representation of Combinator Modes $>$, ML and J

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.

The *polynomial time combinators* assumption in the following theorem 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 constant time for the same reason.

## 5.2 Diagrammatical Parsing

When considering [2] way of using string diagrams for syntactic parsing/reductions, we can see them as (yet) another way of writing our parsing rules. In our typed category, we can make see our combinators as natural transformations (2-cells): then we can see the different sets of combinators as different arity natural transformations. $>$, $\mathrm{ML}_\mathtt{F}$ and $\mathrm{J}_\mathtt{F}$ are represented in Figure 6, up to the coloring of the regions, because that is purely for an artistic rendition.

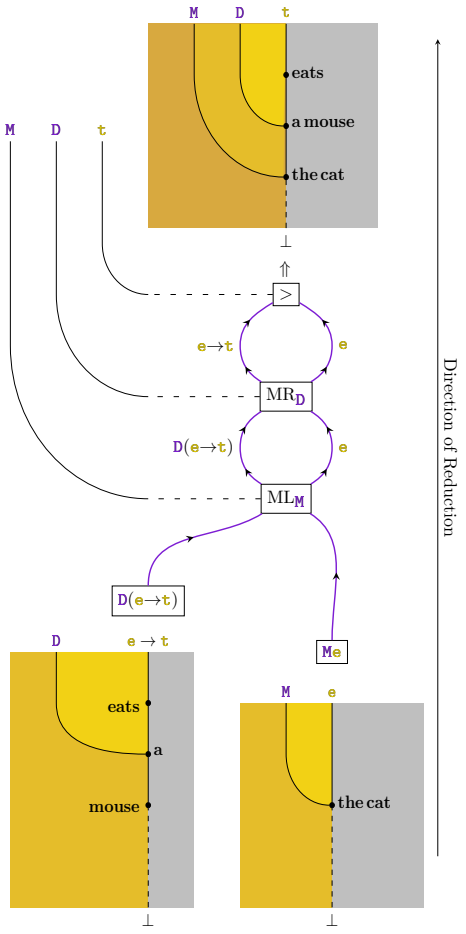



**Figure 5** Representation of a parsing diagram for the sentence *the cat eats a mouse*. See Figure 4b for translation in a parse tree.

Now, a way to see this would be to think of this as an orthogonal set of diagrams 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 2, and then combine the diagrams as shown in Figure 5, which highlights why we can consider the diagrams orthogonal. In this figure we exactly see the sequence of reductions play out on the types of the words, and thus we also see what exact *quasi-braiding* would be needed to construct the effect diagram. Here we talk about *quasi-braiding* because, in a sense, we use 2-cells to do braiding-like operations on the strings, and don't actually allow for braiding inside the diagrammatic computation. To better understand what happens in those parsing diagrams, Figure 4 provides the translations in labelled trees of the parsing

diagrams of Figures 5, 7 and 9.

Categorically, what this means is that we start from a meaning category $\mathcal{C}$, our typing category, and take it as our grammatical category. This is a form of extension on the monoidal version by [2], as it is seemingly a typed version, where we change the pregroup category for the typing category, possibly taken with a product for representation of the English grammar representation, to accommodate for syntactic typing on top of semantic typing. This is, again, just another rewriting of our typing rules.

We have a first axis of string diagrams in the category $\mathcal{C}$ - our string diagrams for effect handling, as in Section 4 - and a second *orthogonal* axis of string diagrams on a larger category, with endofunctors labelled by the types in our typing category $\bar{\mathcal{C}}$ and with natural transformations mapping the combinators defined in Figures 2 and 3. The category in which we consider the second-axis string diagrams does not have a meaning in our compositional semantics theory, and to be more precise, we should talk about 1-cells and 2-cells instead of functors and natural transformations, to keep in the idea that this is really just a diagrammatic way of computing and presenting the operations that are put to work during semantic parsing: A good approach to these diagrams is that they are an extension of the labelled parse trees presented in [1], forming a full diagrammatic calculus system for semantic parsing.

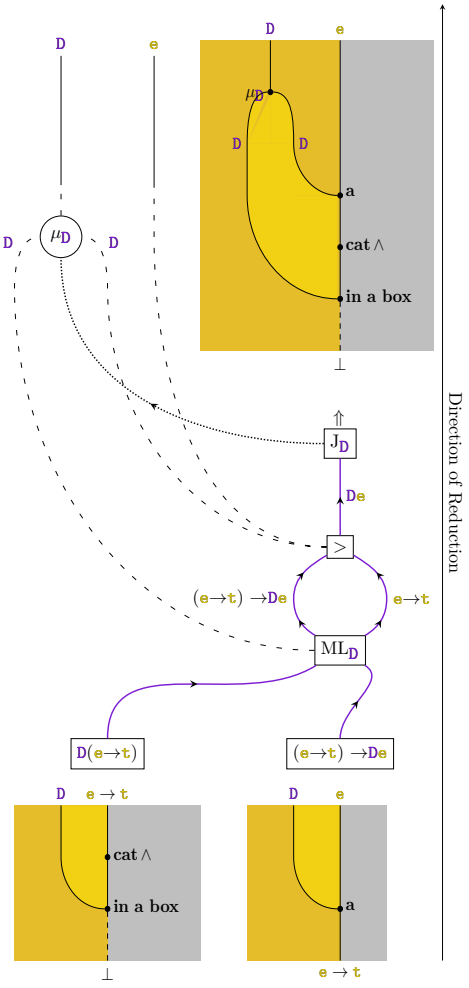



**Figure 7** Example of a parsing diagram for the phrase *a cat in a box*, presenting the integration of unary combinators inside the connector line. See Figure 4b for translation in a parse tree.

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 7. 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.

The main 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 seen before, string diagrams make it easy to compute normal forms, when provided with a confluent reduction system.

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

Suppose you're knitting a rainbow scarf. You have multiple threads (the different words) of the different colours (their types and effects) you're using to knit the scarf. When you decide to change the color you take the different threads you have been using, and mix them up. You can create a new colour[2] 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 9.


**Figure 8** Example of a *Jacquard* knitwork. Photography and work courtesy of the author's mother.

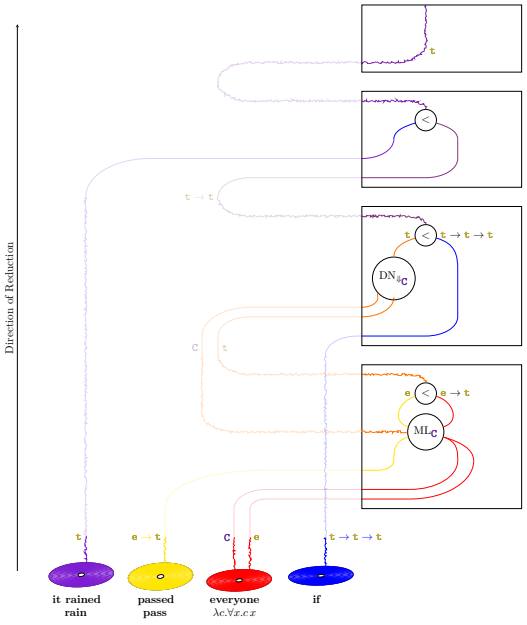



**Figure 9** Knitting-like representation of the diagrammatic parsing of a sentence. See Figure 4c for the translation in a parse tree

The sections in the rectangle represent what happen when considering our combination step as implementing patterns inside a knitwork, as seen in Figure 8. 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 $\mathtt{F}\tau$ and $\mathtt{G}\tau'$. In that case we could either get a result with effects `FG` or with effects `GF` (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.

[2] 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.

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:

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

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

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

▶ **Theorem 6.** *The rules proposed above yield equivalent results.*

**Proof.** For the first point, it is obvious since based on an equality. For the second point: The rules about not using combinators UL and UR come from the notion of handling and granting termination and decidability to our system. The rules about adding J and DN after moving two of the same effect from the same side (i.e. MLML or MRMR) are normalization of Theorem 1. Indeed, in the denotation, 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 t can be done, and that is equivalent to later points because of the elevator equations, or 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`. ■

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

## 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 prove that it is well-founded theoretically, but also that it doesn't come at the cost of comprehensibility or efficiency.

Moreover, while our methods for implementing a type and effect system have been applied to natural language semantics, it could be applied in any language with purely compositional semantics. Of course, there are still improvements that 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. 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-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 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.

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 (10a and 10b) describing the modeling of a subset of the English language in our formalism.

| **Expression** | Type | $\lambda$-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` | Typeclass |
|---|---|---|
| $\mathtt{G}(\tau) = \mathtt{r} \to \tau$ | $\mathtt{G}\varphi\,(x) = \lambda r.\varphi\,(xr)$ | Monad |
| $\mathtt{W}(\tau) = \tau \times \mathtt{t}$ | $\mathtt{W}\varphi\,(\langle a, p\rangle) = \langle \varphi a, p\rangle$ | Monad |
| $\mathtt{S}(\tau) = \{\tau\}$ | $\mathtt{S}\varphi\,(\{x\}) = \{\varphi(x)\}$ | Monad |
| $\mathtt{C}(\tau) = (\tau \to \mathtt{t}) \to \mathtt{t}$ | $\mathtt{C}\varphi\,(x) = \lambda c.x\,(\lambda a.c\,(\varphi a))$ | Monad |
| $\mathtt{D}(\tau) = \mathtt{s} \to \mathtt{S}(\tau \times \mathtt{s})$ | $\mathtt{D}\varphi\,(\lambda s.\,\{\langle x, x +\!\!+ s\rangle \mid px\}) = \lambda s.\,\{\langle \varphi x, \varphi x +\!\!+ s\rangle \mid px\}$ | Monad |
| $\mathtt{M}(\tau) = \tau + \bot$ | $\mathtt{M}\varphi\,(x) = \begin{cases} \varphi\,(x) & \text{if } \Gamma \vdash x : \tau \\ \# & \text{if } \Gamma \vdash x : \# \end{cases}$ | Monad |

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

**Figure 10** Presentation of a lambda-calculus lexicon for the English language