


Dummy title

Matthieu Pierre Boyer ✉ 🏠 

DI ENS, Paris, France

Department of Linguistics, Yale University, USA

Abstract

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Praesent convallis orci arcu, eu mollis dolor. Aliquam eleifend suscipit lacinia. Maecenas quam mi, porta ut lacinia sed, convallis ac dui. Lorem ipsum dolor sit amet, consectetur adipiscing elit. Suspendisse potenti.

2012 ACM Subject Classification Replace ccsdesc macro with valid one

Keywords and phrases Dummy keyword

Digital Object Identifier 10.4230/LIPIcs.CVIT.2016.23

Acknowledgements I want to thank ...

1 Introduction

What is *a chair*? How do I know that *Jupiter, a planet*, is *a planet*? To answer those questions, [2] 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 expressivity 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 phenomenas 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 [2] to get an overview of the linguistic considerations that are the base of the theory.

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

2 Relative Work

This is not the first time a categorical representation of compositional semantics of natural language is proposed, [3] already suggested an approach based on monoidal categories using



© Matthieu P. Boyer;

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. 23:1–23:22

Leibniz International Proceedings in Informatics



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

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, [7] provide a categorical structure based on Hopf algebra and coloured operads to explain their model of syntax. Similarly, [10] provides a modeling of CFGs using coloured operads. Our approach is based on the suggestion that merge in syntax can be done using labels, independant on how it is mathematically modelised.

3 Categorical Semantics of Effects: A Typing System

In this section, we will designate by \mathcal{L} our language, as a set of words (with their semantics) and syntactic rules to combine them semantically. We denote by $\mathcal{O}(\mathcal{L})$ the set of words in the language whose semantic representation is a low-order function and $\mathcal{F}(\mathcal{L})$ the set of words whose semantic representation is a functor or high-order function. Our goals here are to describe more formally, using a categorical vocabulary, the environment in which the typing system for our language will exist, and how we connect words and other linguistic objects to the categorical formulation.

3.1 Typing Category

3.1.1 Types

Let \mathcal{C} be a closed cartesian category, which will be used to represent the domain of types for our domain of denotations. This represents our main typing system, consisting of words $\mathcal{O}(\mathcal{L})$ that can be expressed without effects. Remember that \mathcal{C} contains a terminal object \perp representing the empty type or the lack thereof. We can consider $\bar{\mathcal{C}}$ the category closure of $\mathcal{F}(\mathcal{L})$ ($\mathcal{O}(\mathcal{L})$), that is consisting of all the different type constructors (ergo, functors) that could be formed in the language. What this means is that we consider for our category objects any object that can be attained in a finite number from a finite number of functorial applications from an object of \mathcal{C} . In that sense, $\bar{\mathcal{C}}$ is still a closed cartesian category (since all our functors induce isomorphisms on their image)¹. $\bar{\mathcal{C}}$ will be our typing category (in a way).

We consider for our types the quotient set $\star = \text{Obj}(\bar{\mathcal{C}}) / \mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ does not induce an equivalence relation on $\text{Obj}(\bar{\mathcal{C}})$ but a preorder, we consider the chains obtained by the closure of the relation $x \succeq y \Leftrightarrow \exists F, y = F(x)$ (which is seen as a subtyping relation as proposed in [9]). We also define \star_0 to be the set obtained when considering types which have not yet been *affected*, that is $\text{Obj}(\mathcal{C})$. In contexts of polymorphism, we identify \star_0 to the adequate subset of \star . In this paradigm, constant objects (or results of fully done computations) are functions with type $\perp \rightarrow \tau$ which we will denote directly by $\tau \in \star_0$.

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

¹ Our definition does not yield a closed cartesian category as there are no exponential objects between results of two different functors, but this is not a real issue as we can just say we add those.

3.1.2 Functors, Applicatives and Monads

Our point of view leads us to consider *language functors*² as polymorphic functions: for a (possibly restrained, though it seems to always be \star) set of base types S , a functor is a function

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

Remember that \star is a fibration of the types in $\bar{\mathcal{C}}$. This means that if a functor can be applied to a type, it can also be applied to all *affected* versions of that type, i.e. $\mathcal{F}(L)(\tau \in \star)$. More importantly, while it seems that F 's type is the identity on \star , the important part is that it changes the effects applied to x (or τ). In that sense, F has the following typing judgements:

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

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.

In the same fashion, we can consider functions to have a type in \star or more precisely of the form $\star \rightarrow \star$ which is a subset of \star . This justifies how functors can act on functions in our typing system, thanks to the subtyping judgement introduced above, as this provides a way to ensure proper typing while just propagating the effects. Because of propagation, this also means we can resolve the effects or keep on with the computation at any point during parsing, without any fear that the results may differ.

In that sense, applicatives and monads only provide with more flexibility on the ways to combine functions: they provide intermediate judgements to help with the combination of trees. For example, the multiplication of the monad provides a new *type conversion* judgement:

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

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 also add judgements for adjunctions, but the most interesting thing is to add judgements for natural transformations, as adjunctions are particular examples of natural transformations which arise from *natural* settings. While in general we do not want to find natural transformations, we want to be able to express these in three situations:

1. If we have an adjunction $L \dashv R$, we have natural transformations for $\text{Id}_{\mathcal{C}} \Rightarrow L \circ R$ and $R \circ L \Rightarrow \text{Id}_{\mathcal{C}}$. In particular we get a monad and a comonad from a canonical setting.

² The elements of our language, not the categorical construct.

2. To deal with the resolution of effects, we can map handlers to natural transformations which go from some functor F to the Id functor, allowing for a sequential³ computation of the effects added to the meaning. We will develop a bit more on this idea in Paragraph 3.1.3.1 and 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 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.

Remember that in the Haskell programming language, any polymorphic function is a natural transformation from the first type constructor to the second type constructor, as proved by [13]. 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 [8], the use of handlers as annotations to the syntactic tree of the sentence is an appropriate formalism. This could also give us a way to construct handlers for our effects as per [1], or [11]. As considered by [14] and [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 [12] 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 , $h \circ \eta = \text{id}$.

What does this mean in our typing category? It means that either our language or our parser for the language⁴ should contain natural transformations $F \Rightarrow \text{Id}$ for $F \in \mathcal{F}(\mathcal{L})$. In this goal, we remember that from any polymorphic function in *Haskell* we get a natural transformation [13] meaning that it is enough to be able to define our handler in *Haskell* to be ensured of its good definition. Note that the choice of the handler being part of the lexicon or the parser over the other is a philosophical question more than a semantical one, as both options will result in semantically equivalent models, the only difference will be in the way we consider the resolution of effects. Mathematically⁵, this means the choice of either one of the options is purely of detail left during the implementation.

However, this choice does not arise in the case of the adjunction-induced handlers. Indeed here, the choice is caused by the non-uniqueness of the choices for the handlers. For example, two different speakers may have different ways to resolve the **S** (which will be introduced as the *Powerset* monad in Section ??) that arises from the phrase *A chair*. This usual example of the differences between the cognitive representation of words is actually a specific example of the different possible handlers for the powerset representation of non-determinism/indefinites: there are $|S|$ arrows from the initial object to S in *Set*, representing the different elements of

³ In particular, non-necessarily commutative

⁴ Depending whether we think handling effects is an intrinsic construct of the semantics of the language or whether it is associated with a speaker.

⁵ Computer-wise, actually.

160 *S*. In that sense, while handlers may have a normal form or representation purely dependant
 161 on the effect, the actual handler does not necessarily have a canonical form. This is the
 162 difference with the adjunctions: adjunctions are intrinsic properties of the coexistence of the
 163 effects⁶, while the handlers are user-defined. As such, we choose to say that our handlers
 164 are implemented parser-side but again, this does not change our modelisation of handlers as
 165 natural transformations and most importantly, this does not add non-determinism to our
 166 model: The non-determinism that arises from the variety of possible handlers does not add
 167 to the non-determinism in the parsing.

168 3.1.3.2 Higher-Order Constructs

169 We might want to add plurals, superlatives, tenses, aspects and other similar constructs
 170 which act as function modifiers. For each of these, we give a functor Π corresponding to a
 171 new class of types along with natural transformations for all other functors F which allows
 172 to propagate down the high-order effect. This transformation will need to be from $\Pi \circ F$
 173 to $\Pi \circ F \circ \Pi$ or simply $\Pi \circ F \Rightarrow F \circ \Pi$ depending on the situation. This allows us to add
 174 complexity not in the compositional aspects but in the lexicon aspects.

175 One of the main issues with this is the following: In the English language, plural is
 176 marked on all words (except verbs, and even then case could be made for it to be marked),
 177 while future is marked only on verbs (through the *will + verb* construct which creates a “new”
 178 verb in a sense) though it applies also to the arguments on the verb. A way to solve this
 179 would be to include in the natural transformations rules to propagate the functor depending
 180 on the type of the object. Consider the superlative effect **most**⁷. As it can only be applied on
 181 adjectives, we can assume its argument is a function (but the definition would hold anyway
 182 taking $\tau_1 = \perp$). It is associated with the following function (which is a rewriting of the
 183 natural transformation rule):

$$184 \quad \frac{\Gamma \vdash x : \tau_1 \rightarrow \tau_2}{\Gamma \vdash \mathbf{most} \, x := \Pi_{\tau_2} \circ x = x \circ \Pi_{\tau_1}}$$

185 What currying implies, is that higher-order constructs can be passed down to the arguments
 186 of the functions they are applied to, explaining how we can reconcile the following semantic
 187 equation even if some of the words are not marked properly:

$$188 \quad \mathbf{future}(\mathbf{be}(\mathbf{I}, \mathbf{a \, cat})) = \mathbf{will \, be}(\mathbf{future}(\mathbf{I}), \mathbf{a \, cat}) = \mathbf{be}(\mathbf{future}(\mathbf{I}), \mathbf{a \, cat})$$

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

$$190 \quad \begin{aligned} \mathbf{future}(\mathbf{be})(\mathbf{arg}_1, \mathbf{arg}_2) &= \mathbf{future}(\mathbf{be})(\mathbf{arg}_2)(\mathbf{future}(\mathbf{arg}_1)) \\ &= \mathbf{future}(\mathbf{be})(\mathbf{future}(\mathbf{arg}_2))(\mathbf{future}(\mathbf{arg}_1)) \end{aligned}$$

191 This is not a definitive rule as we could want to stop at a step in the derivation, depending
 192 on our understanding of the notion of future in the language.

193 3.1.3.3 Monad Transformers

194 In [2], the authors present constructions which they call monad transformers or *higher-order*
 195 *constructors* and which take a monad as input and return a monad as output. One way to

⁶ Which we identify to their functorial representations, which we may identify to their free monad in the framework [12].

⁷ We do not care about morphological markings here, we say that **largest** = **most** (**large**)

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

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

196 type those easily would be to simply create, for each such construct, a monad (the result
 197 of the application to any other monad) and a natural transformation which mimics the
 198 application and can be seen as the constructor.

199 3.2 Typing Judgements

200 To complete this section, Figure 1 gives a simple list of different typing composition judgements
 201 through which we also re-derive the subtyping judgement to allow for its implementation.
 202 Note that here, the syntax is not taken into account: a function is always written left of its
 203 arguments, whether or not they are actually in that order in the sentence. This issue will be
 204 resolved by giving the syntactic tree of the sentence (or inferring it at runtime). We could
 205 also add symmetry induced rules for application.

206 Furthermore, note that the **App** rule is the **fmap** rule for the identity functor and that
 207 the **pure/return** and **»=** is the **nat** rule for the monad unit (or applicative unit) and
 208 multiplication.

209 Using these typing rules for our combinators, it is important to see that our grammar
 210 will still be ambiguous and thus our reduction process will be non-deterministic.

211 This non-determinism is a component of our language's grammar and semantics: a same
 212 sentence can have multiple interpretation without context. It is also important to note that
 213 we want to be able to map effects in any possible order. By the same reasoning we applied on
 214 handlers, we choose not to resolve the ambiguity in the parser but we will provide methods
 215 in Sections 4 and 5 to it to a minimum.

216 3.3 Decidability

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

226 4 Handling Non-Determinism

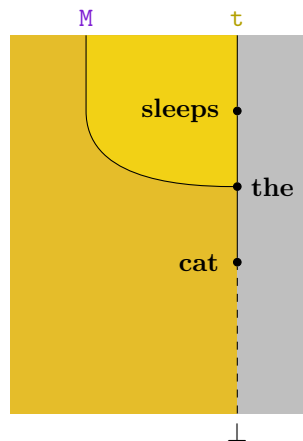
227 The typing judgements proposed in Section 3.2 lead to ambiguity. In this section we propose
 228 ways to get our derivations to a certain normal form, by deriving an equivalence relation on
 229 our derivation and parsing trees, based on string diagrams.

230 4.1 String Diagram Modelisation of Sentences

231 String diagrams are the Poincaré duals of the usual categorical diagrams when considered
 232 in the 2-category of categories and functors. This means that we represent categories as
 233 regions of the plane, functors as lines separating regions and natural transformations as the
 234 intersection points between two lines. We will always consider application as applying to
 235 the right of the line so that composition is written in the same way as in equations. This
 236 gives us a new graphical formalism to represent our effects using a few equality rules between
 237 diagrams. The commutative aspect of functional diagrams is now replaced by an equality of
 238 string diagrams, which will be detailed in the following section.

239 The important aspect of string diagrams that we will use is that two diagrams that are
 240 planarily homotopic are equal [6] and that we can map a string diagram to a sequence of
 241 computations on computation: the vertical composition of natural transformations (bottom-
 242 up) represents the reductions that we can do on our set of effects. This means that even
 243 when adding handlers, we get a way to visually see the meaning get reduced from effectful
 244 composition to propositional values, without the need to specify what the handler does.
 245 Indeed we only look at *when* we apply handlers and natural transformations reducing the
 246 number of effects. This delimits our usage of string diagrams as ways to look at computations
 247 and a tool to provide equality rules to reduce non-determinism by constructing an equivalence
 248 relationship (which we denote by \equiv) and yielding a quotient set of *normal forms* for our
 249 computations.

250 Let us define the category $\mathbf{1}$ with exactly one object and one arrow: the identity on that
 251 object. It will be shown in grey in the string diagrams below. A functor of type $\mathbf{1} \rightarrow \mathcal{C}$
 252 is equivalent to choosing an object in \mathcal{C} , and a natural between two such functors τ_1, τ_2 is
 253 exactly an arrow in \mathcal{C} of type $\tau_1 \rightarrow \tau_2$. Knowing that allows us to represent the type resulting
 254 from a sequence of computations as a sequence of strings whose farthest right represents an
 255 object in \mathcal{C} , that is, a base type.



256

For simplicity reasons, and because the effects that are buried in our typing system not only give rise to functors but also have types that are not purely curriffiable, we will write our string diagrams on the fully parsed sentence, with its most simplified/composed expression. Indeed, the question of providing rules to compose the string diagram for **the** and the one for **cat** to give the one for **the cat** is a difficult question, that natural solutions to are not obvious, and will be discussed in the next section.

To justify our proposition to only consider fully reduced expressions note that in this formalism we don't consider the expressions for our functors and natural transformations but simply the sequence in which they are applied. This works since the following diagrams commute for any F, G functors and θ natural transformation:

$$\begin{array}{ccc} G & \xrightarrow{F} & F \circ G \\ \theta \downarrow & & \downarrow F \circ \theta \\ \theta G & \xrightarrow{F} & F \circ \theta G \end{array} \qquad \begin{array}{ccc} G & \xrightarrow{F} & G \circ F \\ \theta \downarrow & & \downarrow \theta \circ F \\ \theta G & \xrightarrow{F} & \theta G \circ F \end{array}$$

The property of natural transformations to be applied before or after any arrow in the category justifies that even when composing before the handling we get the same result. These properties justifies the fact that we only need to prove the equality of two reductions at the farthest step of the reductions, even though in practice the handling might be done at earlier points in the parsing.

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.

There is however one thing that may seem to be an issue: existential quantifiers inside **if then** sentences and other functions looking like $\text{Ma} \rightarrow \text{b}$. Indeed, [2] suggests that those could be of type $\text{St} \rightarrow \text{t}$ and that we might want to apply that function to something of type St by first using the unit of the monad S then invoking fmap . There is no issue with that in our typing system, as this combination mode is the composition of two typing judgements (in a sense). To graphically reconcile this and our string diagrams, especially since units and handlers cancel each other, we suggest to see the effects as being dragged along the parsing trees (or parsing string diagrams) until they're never needed again.

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 [6] 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 [6]). *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 Elevator Equations are a consequence of Theorem 1 but also highlight one of the most important properties of string diagrams in their modelisation of multi-threaded computations: what happens on one string does not influence what happens on another in the same time.

The Snake Equations are a rewriting of the categorical diagrams 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.

[4] 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 [4] in their combinatorial description of string diagrams. We describe a diagram by an ordered set of its 2-cells (the natural transformations) along with the number of input strings, for each 2-cell the following information:

- Its horizontal position: the number of strings that are right of it (we adopt this convention to match our graphical representation of effects: the number of strings is the distance to the base type).
- Its type: an array of effects (read from left to right) that are the inputs to the natural transformation and an array of effects that are the outputs to the natural transformation. The empty array represents the identity functor. Of course, we will not actually copy arrays and arrays inside our datastructure but simply copy labels which are keys to a dictionary containing such arrays to limit the size of our structure, allowing for $\mathcal{O}(1)$ access to the associated properties.

We will then write a diagram D as a tuple of 3 elements⁸: $(D.N, D.S, D.L)$ 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 $nat = (nat.h, nat.in, nat.out)$. 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 [4]. This in particular means that our structure can be normalized in polynomial time to check for equality under Theorem 1. More precisely, the complexity of our algorithm is in $\mathcal{O}(n \times \sup_i |D.L(i).in| + |D.L(i).out|)$, 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. 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 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:

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

⁸ We could write it as a tuple of 5 elements by replacing $D.L$ by three functions that lower level

| | | | |
|---|--|--|--|
| $>, \beta$ | $::= (\alpha \rightarrow \beta), \alpha$ | $ML_F(\alpha, \beta)$ | $::= F\alpha, \beta$ |
| $<, \beta$ | $::= \alpha, (\alpha \rightarrow \beta)$ | $MR_F(\alpha, \beta)$ | $::= \alpha, F\beta$ |
| $\wedge, \alpha \rightarrow \mathbf{t}$ | $::= (\alpha \rightarrow \mathbf{t}), (\alpha \rightarrow \mathbf{t})$ | $A_F(\alpha, \beta)$ | $::= F\alpha, F\beta$ |
| $\vee, \alpha \rightarrow \mathbf{t}$ | $::= (\alpha \rightarrow \mathbf{t}), (\alpha \rightarrow \mathbf{t})$ | $UR_F(\alpha \rightarrow \alpha', \beta)$ | $::= F\alpha \rightarrow \alpha', \beta$ |
| | | $UL_F(\alpha, \beta \rightarrow \beta')$ | $::= \alpha, F\beta \rightarrow \beta'$ |
| $J_F F\tau$ | $::= FF\tau$ | $C_{LR}(L\alpha, R\beta)$ | $::= (\alpha, \beta)$ |
| $DN_C \tau$ | $::= C_\tau \tau$ | $ER_R(R(\alpha \rightarrow \alpha'), \beta)$ | $::= \alpha \rightarrow R\alpha', \beta$ |
| | | $EL_R(\alpha, R(\beta \rightarrow \beta'))$ | $::= \alpha, \beta \rightarrow R\beta'$ |

■ **Figure 2** Possible Type Combinations in the form of a near CFG

as it simplifies the rewriting of our typing judgements in a CFG⁹. The grammars provided in Figures ?? and 2 are actually used from right to left, as we actually do combinations over the types and syntactic categories of the words and try to reduce the sentence, not derive it from nothing. Now, all the improvements discussed in Section 5.3 can still be applied here, just a bit differently, as this amounts to reducing ambiguity in the grammar.

This grammar works in five major sections:

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

The idea of the *grammatical* reduction is that from the flat sentence, we create a full parse tree as a sequence of types τ . We then reduce it using the binary effect combinators, before choosing the appropriate binary type combinator. It is at this point in the reduction we actually do the composition of functions. We close up the reduction¹⁰ by possibly using unary effect combinators.

We do not prove here that these typing rules cannot be written in a simpler syntactic structure¹¹, but it is easy to see why a regular expression would not work, and since we need trees, to express our full system, the best we can do would be to disambiguate the grammar. A thing to note is that this grammar is not complete but explains how types can be combined in a compact form. For example, the rules of the form $ML_F M, \tau' \leftarrow M, \tau$ are rules that provide ways to combine effects from the two inputs in the order we want to: we can combine $RS\mathbf{e}$ and $CW(\mathbf{e} \rightarrow \mathbf{t})$ into $RCWS\mathbf{t}$ with the mode $MLMRMRML <$ (see [2] Example 5.14).

Each of these combinators can be, up to order, associated with a inference rule, and, as such, with a higher-order denotation, which explains the actual effect¹² of the combinator, and are described in Figure 3. The main reason we need to get denotations associated

⁹ That, in a sense, was already implicitly provided in Figure 1.

¹⁰ For the combination of two words, which is then repeated along the syntactic structure.

¹¹ It is important to note that for a typing system, we only have syntactic-like rules that allow, or not, to combine types.

¹² Pun intended

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

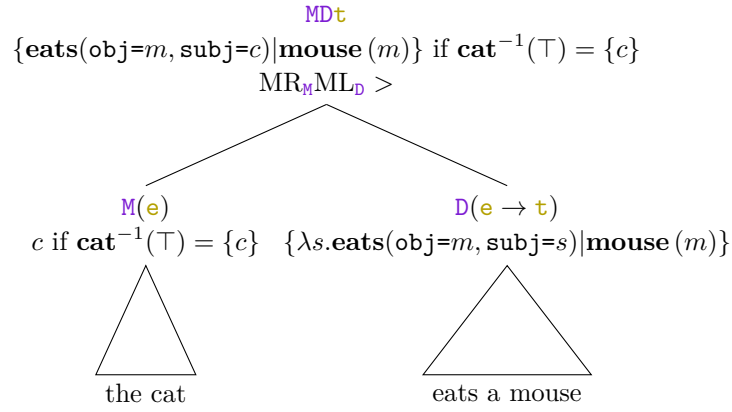
■ **Figure 3** Denotations describing the effect of the combinators used in the grammar describing our combination modes presented in Figure 2

to combinators, is to properly define equivalence and reduce the number of rules in the grammar. Explanation on how that is done will come in Section 5.3. 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. The same structure for combinators apply when describing the combinators than when describing their rules of existence. This makes us able to compute the actual denotations associated to a sentence using our formalism, as presented in figure 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.

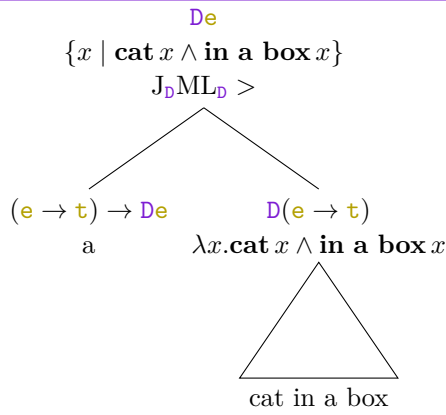
Moreover, it is important to note that while we talk about the structure in Figure 2 as a grammar, it is more of a structure that computes if it is possible or not to combine two types and how. In particular, our grammar is not finite, since there are infinitely many types that can be playing the part of α and β , and infinitely many functors that can play the roles for \mathbf{F} and so on, but that does not pose a problem, as recognizing the general form of a type can be done in constant time for the forms that we want to check, given a proper memory representation. Furthermore, it can easily be rendered into something that looks like a Chomsky Normal Form for a CFG and allows us to integrate this in the CYK algorithm, and even get a correct time complexity. Since our grammar is not actually finite, the modified version of CYK that parses only the types (not the full system) will have a complexity in the size of $\mathcal{F}(\mathcal{L})$ that is linear, albeit with a not so small constant factor, which comes from the fact that our grammar's size is linear in $|\mathcal{F}(\mathcal{L})|$.

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

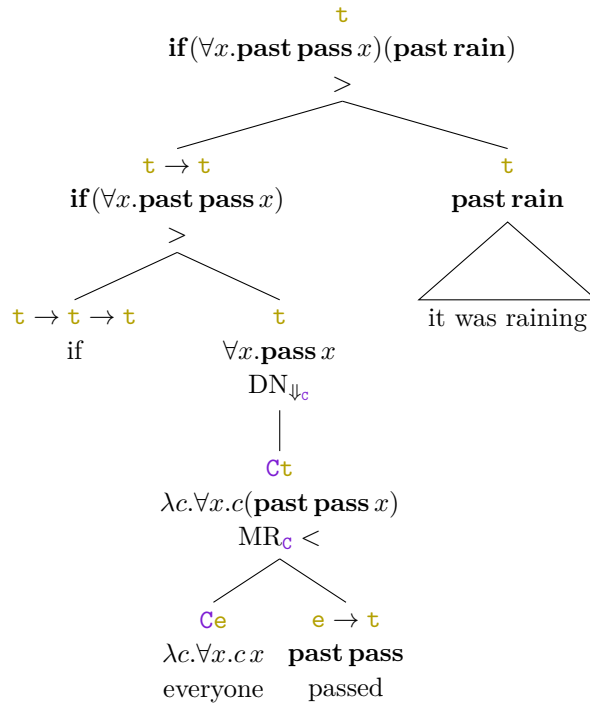
Proof. Suppose we are given a syntactic generating structure G_s along with our type combination grammar G_τ . The system G that can be constructed from the (tensor) product of G_s and G_τ has size $|G_s| \times |G_\tau|$. Indeed, we think of its rules as a rule for the syntactic categories and a rule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



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



(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.

input and in $|G_s|$, then we can syntactico-semantically parse it in polynomial time in the size of the input, $|G_s|$ and $|G_\tau|$. ■

While we have gone with the hypothesis that we have a CFG for our language, any type of polynomial-time structure could work¹³, as long as it is more expressive than a CFG. We will do the following analysis using a CFG since it allows to cover enough of the language for our example and for simplicity of describing the process of adding the typing CFG, even though some think that English is not a context free language [5]. Since the CYK algorithm provides us with an algorithm for CFG based parsing cubic in the input and linear in the grammar size, we end up with an algorithm for semantically parsing a sentence that is cubic in the length of the sentence and linear in the size of the grammar modeling the language and the type system.

► **Theorem 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. ■

The *polynomial time combinators* assumption is not a complex assumption, this is for example true for our denotations in Section ??, with function application being linear in the number of uses of variables in the function term, which in turn is linear in the number of terms used to construct the function term and thus of words, and `fmap` being in constant time¹⁴ for the same reason. Of course, function application could be said to be in constant time too, but we consider here the duration of the β -reduction:

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

5.2 Diagrammatic Parsing

When considering [3] way of using string diagrams for syntactic parsing/reductions, we can see them as (yet) another way of writing our parsing rules. In our typed category, we can make see our combinators as natural transformations (2-cells): then we can see the different sets of combinators as different arity natural transformations. $>$, ML_F and J_F are represented in Figure 5, up to the coloring of the regions, because that is purely for an artistic rendition¹⁵.

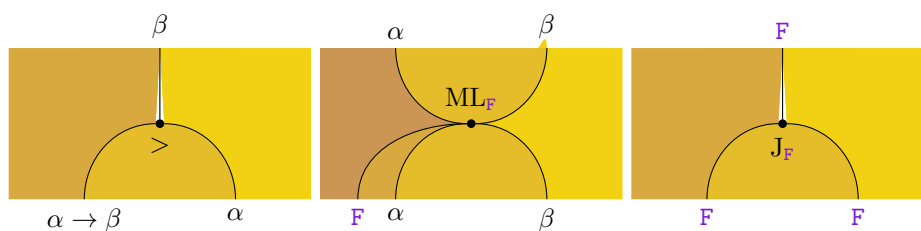
Now, a way to see this would be to think of this as an orthogonal set of diagrams to the ones of Section 4: we could use the syntactic version of the diagrams to model our parsing, according to the rules in Figure 2, and then combine the diagrams as in Equation ??, as shown in Figure 6, which highlights why we can consider the diagrams orthogonal. The obvious reaction¹⁶ is: "Why did we start by saying we could just paste?". This is a perfectly

¹³Simply, the algorithm will need to be adapted to at least process the CFG for the typing rules.

¹⁴Depending on the functor but still bounded by a constant.

¹⁵Colours make this less boring, trust me

¹⁶That I'm sure everyone have had seeing this.



■ **Figure 5** String Diagrammatic Representation of Combinator Modes $>$, ML and J

■ **Figure 6** Representation of a parsing (sub-)diagram for the sentence *the cat eats a mouse*, including the connection between the effect handling diagrams from Section 4 and the syntactio-semantic parsing diagrams formed combining our approach to parsing and [3] approach to diagrams. See Figure 4b for translation in a parse tree.

valid point, and pasting diagrams is simply a notation abuse, that is justified by the fact there is a possible reduction. In this figure we exactly see the 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-like¹⁷ 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 6, 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 [3], as it is seemingly a typed version, where we change the Pregroup category for the typing category, possibly taken with a product for representation of the English grammar representation, to accomodate for syntactic typing on top of semantic typing. This is, again, just another rewriting of our typing rules.

More formally, 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 endofunctors¹⁸ graded 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.

What the lines leading from combinators to functors¹⁹ mean categorically, is void in either category. Those lines are not actually part of the first axis of the string diagram, nor are they part of the second axis of the string diagram. They are used to map out the link between the two: they express the quasi-braiding step proposed above, and present graphically why the order of the strings in the resulting diagram is as it is, and what the pasting and quasi-braiding orders should be. A good approach to these diagrams might

¹⁷Permutations on the order of the strings

¹⁸To ensure proper typing of the diagrams.

¹⁹Remember that types are objects in \mathcal{C} and thus functors from $\mathbf{1} \Rightarrow \mathcal{C}$

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



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

542 be the following: they are an extension of the parse trees presented in [2], applied to the
 543 formalism of [3] and extended to be integrated with the handling of effects, as described in
 544 Section 4, forming a full diagrammatic calculus system for semantic parsing.

545 For the combinators J, DN and C, which are applied to reduce the number of effects
 546 inside a denotation, it might seem less obvious how to include them. While this may seem
 547 true, it's actually only true for the *connecting* strings. Indeed, applying them to the actual
 548 *parsing* part of the diagram is done in the exact same way as in the CFG, we just apply
 549 them when needed, and they will appear in the resulting denotation as an end for a string, a
 550 form of forced handling, in a sense, as shown in the result of Figure 7. For the connecting
 551 strings, it's simply a matter of adding a new *phantom* string that will send the associated
 552 2-cell in the effect handling diagram to the connected strings. In particular it is interesting
 553 to note that the resulting diagram representing the sentence can, in a way, be found in the
 554 connection strings that arise from the combinators.

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

559 The other reason being the tangible interpretation of how things work underlying the
 560 idea of a string diagram: Suppose you're knitting a rainbow scarf. You have multiple threads
 561 (the different words) of the different colours (their types and effects) you're using to knit
 562 the scarf. When you decide to change the color you take the different threads you have
 563 been using, and mix them up: You can create a new colour²⁰ thread from two (that's the
 564 base combinators) create a thicker one from two of the same colour (the applicative mode

²⁰ I know this is not how wool works, but if you prefer you can imagine a pointillist-like way of drawing

■ **Figure 9** 3D-like Representation of the Diagrammatic Parsing of a Sentence. See 4c for the translation in a Parse Tree

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. 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.

5.2.1 The Coproduct Categorical Approach to Syntax

In this section based on the work by [7], we explore the integration of our notion inside the structure of syntactic merge, and why the two formalisms are compatible. The idea behind that structure is to see the union of trees as a product and the merging of trees as based on a coproduct in a well-defined Hopf algebra. Now of course, with our insights on syntactico-semantic parsing, we can either think of our parsing structure labeled by multiple modes, as in [2], or think of it as string diagrams, as presented above. In both cases, we make a more or less implicit use of the merge operation.

Labeled trees can be seen as computed from a sequence of labeled merge. As such, it is easy to see how one can adapt the construction of a purely syntactic merge onto a semantic merge, using a similar approach. There is just the need for a coloured operad to create a series of labeled merges, like proposed in [10] for example. Since there is a one-to-one mapping from our string diagrams to parse trees, mapping explained when comparing the tables for denotations of combinators in Figure 3 and Figure 12 in [2], or by looking at the parse trees equivalent to string diagrams in Figures 6, 7 and 9, it is easy to see that indeed, there is a notion of merge inside our diagrams that can in a way be expressed through a Hopf algebra, by transporting the diagrams to and from their equivalent parsing diagram. More generally, what this means is that a merge, in our string diagrams, is the addition of a combinator to one, two²¹ without conditions. This is useful as a definition, because it allows integration of our system in a broader framework, including for example the notion of morphosyntactic trees.

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

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.

²¹ Or more, see ??

5.3 Rewriting Rules

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

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

First, let's consider the reductions proposed in Section 4. Those define normal forms under Theorem 2 and thus we can use those to reduce the number of trees. Indeed, we usually want to reduce the number of effects by first using the rules *inside* the language, understand, the adjunction co-unit and monadic join. Thus, when we have a presupposition of the form:

$$\frac{}{\Gamma \vdash x : \text{MM}\tau} \quad \top \quad \text{or} \quad \frac{}{\Gamma \vdash x : \text{RL}\tau} \quad \top$$

we always simplify it directly, as not always doing it would propagate multiple trees. This in turn means we always verify the derivational rules we set up in the previous section for the join equations of the monad.

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

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

Indeed, in that case we could either get a result with effects FG or with effects GF . In general those effects are not the same, but if they happen to be, which trivially will be the case when one of the effects is external, the plural or islands functors for example. When the two functors commute, the order of application does not matter and thus we choose to get the outer effect the one of the left side of the combinator.

Thirdly, there are modes that clearly encompass other ones²². One should not use mode UR when using MR or DNMR and the same goes for the left side, because the two derivations yield simpler results. Same things can be said for certain other derivations containing the lowering and co-unit combinators.

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

²² Here we use the grammar notation for ease of explanation

| | | | | | |
|-----|---|---|-----|---|-----------------|
| 638 | ■ | m_F, DN, m_F where $m \in \{MR, ML\}$ | 641 | ■ | ML_F, DN, A_F |
| 639 | ■ | ML_F, DN, MR_F | | | |
| 640 | ■ | A_F, DN, MR_F | 642 | ■ | C |

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

| | | | | | |
|-----|---|--|-----|---|-----------------------------------|
| 644 | ■ | $\{m_F, j, m_F\}$ where $m \in \{MR, ML\}$ | 649 | ■ | If F is commutative as a monad: |
| 645 | ■ | ML_F, j, MR_F | 650 | ■ | MR_F, A_F |
| 646 | ■ | $A_F, j, MR_F,$ | 651 | ■ | A_F, ML_F |
| 647 | ■ | ML_F, j, A_F | 652 | ■ | MR_F, j, ML_F |
| 648 | ■ | k, C for $k \in \{\varepsilon, A_F\}$ | 653 | ■ | A_F, j, A_F |

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

655 **Proof.** For the first point, the equivalence has already been proved under Theorem 2. For
 656 the second point, it is obvious since based on an equality. For the third point, it's a bit more
 657 complicated. The rules about not using combinators UL and UR come from the notion of
 658 handling and granting termination and decidability to our system. The rules about adding
 659 J and DN after moving two of the same effect from the same side (i.e. MLML or MRMR)
 660 are normalization of a the elevator equations ???. Indeed, in the denotation, the only reason
 661 to keep two of the same effects and not join them²³ is to at some point have something get
 662 in between the two. Joining and cloture should then be done at earliest point in parsing
 663 where it can be done, and that is equivalent to later points because of the elevator equations,
 664 or Theorem 1. The last set of rules follows from the following: we should not use JMLMR
 665 instead of A, as those are equivalent because of the equation defining them. The same thing
 666 goes for the other two, as we should use the units of monads over applicative rules and
 667 fmap. ■

668 The observant reader might have noticed that this is simply a scheme to apply typing
 669 rules to a syntactic derivation, but that this will not be enough to actually gain all reductions
 670 possible in polynomial time. This is actually not feasible (because of the intrinsic ambiguity
 671 of the English language, as proved for example by the sentence *The man sees the girl using a*
 672 *telescope*). Even then, we are far from reducing to a minimum the number of different paths
 673 possible to get a same final denotation. This can only happen once a confluent reduction
 674 scheme is provided for the denotations. When this is done, we can combine the reduction
 675 schemes for effects along with the one for denotation and the one for combinations in one
 676 large reduction scheme. Indeed, trivially, the tensor product of confluent reduction schemes
 677 forms a confluent reduction scheme, whose maximal reduction length is the sum of the
 678 maximal reduction lengths²⁴.

679 When using our diagrammatic approach to parsing, which, again, is just a rewriting in
 680 a more graphical fashion of our typing rules and our CFG-like rules, we can write all the
 681 reductions described above to our paradigm: it simply amounts to constructing a set of
 682 normal forms for the string diagrams. This leads to the same algorithms developed in Section
 683 4 being usable here: we just have a new improved version of Theorem 2 which adds the
 684 normal forms specified in this section to the newly added *orthogonal* axis of diagrammatic

²³ Provided they can be joined.

²⁴ Actually it is at most that, but that does not really matter.

computations. What we're actually doing is computing two different normal forms along the tensor product of our reduction schemes²⁵, but again, that only amounts to computing a larger normal form. Moreover, considering the possible normal forms of syntactic reductions simply adds another way to reduce our diagrams to normal forms. Since all of these forms can be attained in polynomial time, it is clear that finding a normal-form diagram, which is exactly a normal-form denotation for the sentence, is doable in polynomial time²⁶

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

References

- 1 Andrej Bauer and Matija Pretnar. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4, December 2014. doi:10.2168/LMCS-10(4:9)2014.
- 2 Dylan Bumford and Simon Charlow. Effect-driven interpretation: Functors for natural language composition, March 2025.
- 3 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.
- 4 Antonin Delpuch and Jamie Vicary. Normalization for planar string diagrams and a quadratic equivalence algorithm, January 2022. arXiv:1804.07832, doi:10.48550/arXiv.1804.07832.
- 5 James Higginbotham. English Is Not a Context-Free Language. *Linguistic Inquiry*, 15(2):225–234, 1984. arXiv:4178381.
- 6 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.
- 7 Marcolli, Matilde et Chomsky, Noam et Berwick, Robert C. Mathematical Structure of Syntactic Merge.
- 8 Jiří Maršík and Maxime Amblard. Algebraic Effects and Handlers in Natural Language Interpretation.
- 9 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.
- 10 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.
- 11 Gordon D. Plotkin and Matija Pretnar. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4, December 2013. doi:10.2168/LMCS-9(4:23)2013.
- 12 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.
- 13 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.

²⁵ Just like we already have (or should have) a tensor scheme for the denotations and combination modes.

²⁶ Of course, this is only true when the denotations can be normalized in polynomial time.

23:22 Dummy title

- 729 **14** Nicolas Wu, Tom Schrijvers, and Ralf Hinze. Effect handlers in scope. In *Proceedings of the*
730 *2014 ACM SIGPLAN Symposium on Haskell*, Haskell '14, pages 1–12, New York, NY, USA,
731 September 2014. Association for Computing Machinery. doi:10.1145/2633357.2633358.

732 **A** Presenting a Language