Formalizing Typing Rules for Natural Languages using Effects

Matthieu Boyer

March 27, 2025

Contents

Abstract

The main idea is that you can consider some words to act as functors in a typing category, for example determiners: they don't change the way a word acts in a sentence, but more the setting in which the word works by adding an effect to the work. This document is based mostly on Bumford and Charlow and Bumford.

Introduction

Moggi (1989) provided a way to view monads as effects in functional programming. This allows for new modes of combination in a compositional semantic formalism, and provides a way to model words which usually raise problems with the traditional lambda-calculus representation of the words. In particular we consider words such as *the* or *a* whose application to common nouns results in types that should be used in similar situations but with largely different

semantics, and model those as functions whose application yields an effect. This allows us to develop typing judgements and an extended typing system for compositional semantics of natural languages. Type-driven compositional semantics acts under the premise that given a set of words and their denotations, a set of grammar rules for composition and their associated typing judgements, we are able to form an enhanced parser for natural language which provides a mathematical representation of the meaning of sentences, as proposed by Heim and Kratzer (1998)

1 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.

1.1 Typing Category

1.1.1 Types

Let $\mathcal{C}$ be a closed cartesian category. 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 $\star = \text{Obj}\left(\bar{\mathcal{C}}\right)/\mathcal{F}(\mathcal{L})$. Since $\mathcal{F}(\mathcal{L})$ does not induce an equivalence relation on $\text{Obj}\left(\bar{\mathcal{C}}\right)$ but a preorder, we consider the chains obtained by the closure of the relation $x \succeq y \Leftrightarrow \exists F, y = F(x)$ (which is seen as a subtyping relation as proposed in Melliès and Zeilberger (2015)). 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 $\perp \to \tau$ 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 types $F(A \Rightarrow B)$ and $FA \Rightarrow FB$.

We will provide in Table 2 a list of the effect-less usual types associated to regular linguistic objects.

1.1.2 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

[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.

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

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:

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

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

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

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

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

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

1.1.3 Natural Transformations for Handlers and Higher-Order Constructs

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

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

2. To deal with the resolution of effects, we can map handlers to natural transformations which go from some functor F to the Id functor, allowing for a sequential[3] computation of the effects added to the meaning. We will develop a bit more on this idea in 1.1.3.2.

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 1.1.3.3.

To see why we want this rule, which is a larger version of the monad multiplication and the monad/applicative unit, it suffices to see that the diagram below provides a way to construct the "correct function" on the "correct functor" side of types. If we have a natural transformation $F \overset{\theta}{\Longrightarrow} G$ then for all arrows $f : \tau_1 \to \tau_2$ we have:

$$\begin{array}{ccc} F\tau_1 & \xrightarrow{Ff} & F\tau_2 \\ {\scriptstyle \theta_{\tau_1}}\downarrow & & \downarrow{\scriptstyle \theta_{\tau_2}} \\ G\tau_1 & \xrightarrow{Gf} & G\tau_2 \end{array}$$

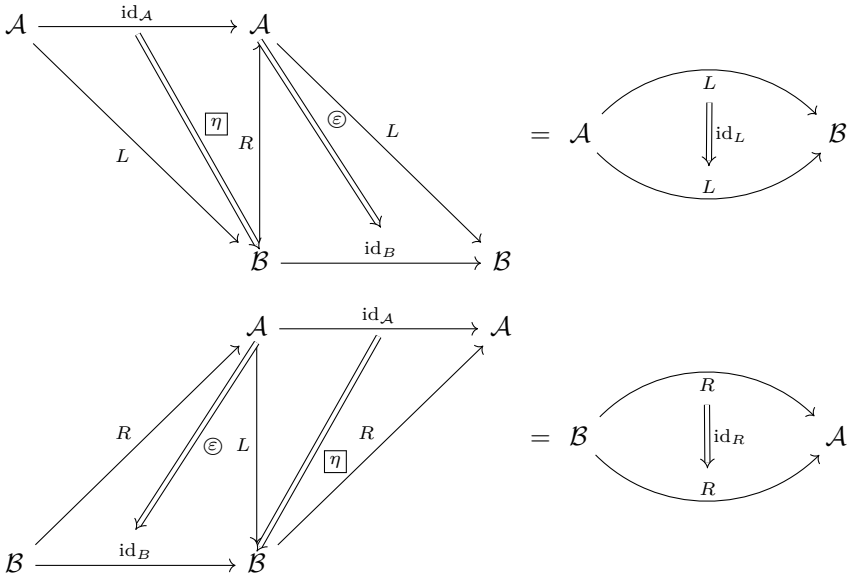
and this implies, from it being true for all arrows, that from $\Gamma \vdash x : F\tau$ we have an easy construct to show $\Gamma \vdash x : G\tau$.

[3] In particular, non-necessarily commutative

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 Wadler (1989). This will guarantee for us that given a *Haskell* construction for a polymorphic function, we will get the associated natural transformation.

1.1.3.1 Adjunctions

We will not go in much details about adjunctions, as a full example is provided in 2.3. To understand why adjunctions are useful, one simply needs to understand that an adjunction $L \dashv R$ is a pair of functors $L : \mathcal{A} \to \mathcal{B}$ and $R : \mathcal{B} \to \mathcal{A}$, and a pair of natural transformations $\eta : \mathrm{Id}_{\mathcal{A}} \Rightarrow R \circ L$ and $\varepsilon : L \circ R \Rightarrow \mathrm{Id}_{\mathcal{A}}$ such that the two following equations are satisfied:



Remember that an adjunction defines two different structures over itself: a monad $L \circ R$ and a comonad $R \circ L$. The fact these structures arise from the interaction between two effects renders them an intrinsic property of the language.

1.1.3.2 Handlers

As introduce by Maršík and Amblard, 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 Bauer and Pretnar (2014), or Plotkin and Pretnar (2013). As considered by Wu et al. (2014) and van den Berg and Schrijvers (2024), 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 (van den Berg and Schrijvers, 2024) we simply need to create handlers for our effects/functors and we will then have in our language the result needed.

What does this mean in our typing category ? It means that either our language or our parser for the language [4] should contain natural transformations $F \Rightarrow \mathrm{Id}$ for $F \in \mathcal{F}(\mathcal{L})$. However, consider the following sentence:

Jupiter, a planet, has a moon.

Here we *record* in memory that Jupiter is a planet, but we never access this record. This leads to the main issue with handling: some effects are meant to be considered at the scale of a text and we might want to consider that sentences are true and be able to use this to provide further reduction and make use of adjunctions instead of *out of the blue* natural transformations.

[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.

1.1.3.3 Higher-Order Constructs

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

One of the main issues with this is the following: In the English language, plural is marked on all words (except verbs, and even then case could be made for it to be marked), while future is marked only on verbs (through the *will + verb* construct which creates a "*new*" verb in a sense) though it applies also to the arguments on the verb. A way to solve this would be to include in the natural transformations rules to propagate the functor depending on the type of the object. Consider the superlative effect **most**[5]. As it can only be applied on adjectives, we can assume its argument is a function (but the definition would hold anyway taking $\tau_1 = \bot$). It is associated with the following function (which is a rewriting of the natural transformation rule):

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

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

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

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

$$\mathbf{future}\,(\mathbf{be})\,(\mathbf{arg_1}, \mathbf{arg_2}) = \mathbf{future}\,(\mathbf{be})\,(\mathbf{arg_2})\,(\mathbf{future}\,(\mathbf{arg_1})) = \mathbf{future}\,(\mathbf{be})\,(\mathbf{future}\,(\mathbf{arg_2}))\,(\mathbf{future}\,(\mathbf{arg_1}))$$

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

1.2 Typing Judgements

To complete this section, Table 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. This issue will be resolved by giving the syntactic tree of the sentence (or infering it at runtime). We could also add symmetry induced rules for application.

Furthermore, note that the App rule is the `fmap` rule for the identity functor and that the `pure/return` and `>>=` is the `nat` rule for the monad unit (or applicative unit) and multiplication.

2 Language Translation

In this section we will give a list of words along with a way to express them as either arrows or endo-functors of our typing category. This will also give a set of functors and constructs in our language.

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

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

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

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

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

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

$$\frac{\Gamma \vdash x : \tau}{\Gamma \vdash x : A\tau} \quad \texttt{pure/return}$$

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

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

Table 1: Typing and Subtyping Judgements

2.1 Types of Syntax

We first need to setup some guidelines for our denotations, by asking ourselves how the different components of sentences interact with each other. For syntactic categories, the types that are generally used are the following, and we will see it matches with our lexicon, and simplifies our functorial definitions[6]. Those are based on Partee. Here Υ is the operator which retrieves the type from a certain syntactic category.

2.2 Lexicon: Semantic Denotations for Words

Many words will have basically the same "high-level" denotation. For example, the denotation for most common nouns will be of the form: $\Gamma \vdash \lambda x.\textbf{planet}x : \texttt{e} \to \texttt{t}$. In Table 3 we give a lexicon for a subset of the english language. We describe the constructor for the functors used by our denotations in the table, but all functors will be reminded and properly defined in Table 4 along with their respective `fmap`.

2.3 Effects of the Language

For the applicatives/monads in Table 4 we do not specify the unit and multiplication functions, as they are quite usual examples. We still provide the `fmap` for good measure.

Let us explain a few of those functors: `G` designates reading from a certain environment of type `r` while `W` encodes the possibility of logging a message of type `t` along with the expression. The functor `M` describes the possibility for a computation to fail, for example when retrieving a value that does not exist (see **the**). The `S` functor represents the space of possibilities that arises from a non-deterministic computation (see **which**).

[6] We don't consider effects in the given typings.

S $\mathtt{t}$

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

ProperN $\mathtt{e}$

NP Either:

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

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

ADJ(P) Either:

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

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

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

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

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

Table 2: Usual Typings for some Syntactic Categories

Expression	Type	λ-Term
planet	$\mathtt{e} \to \mathtt{t}$	$\lambda x.\mathbf{planet}\, x$
	Generalizes to **common nouns**	
carnivorous	$(\mathtt{e} \to \mathtt{t})$	$\lambda x.\mathbf{carnivorous}\, x$
	Generalizes to **predicative adjectives**	
skillful	$(\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t})$	$\lambda p.\lambda x.px \wedge \mathbf{skillful}\, x$
	Generalizes to **predicate modifier adjectives**	
Jupiter	$\mathtt{e}$	$\mathbf{j} \in \mathrm{Var}$
	Generalizes to **proper nouns**	
sleep	$\mathtt{e} \to \mathtt{t}$	$\lambda x.\mathbf{sleep}\, x$
	Generalizes to **intranitive verbs**	
chase	$\mathtt{e} \to \mathtt{e} \to \mathtt{t}$	$\lambda x.\lambda y.\mathbf{chase}\,(x, y)$
	Generalizes to **transitive verbs**	
be	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{e} \to \mathtt{t}$	$\lambda p.\lambda x.px$
she	$\mathtt{e} \to \mathtt{e}$	$\lambda x.x$
it	$(\bot \to \mathtt{Ge})$	$\lambda g.g_0$
which	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Se}$	$\lambda p.\,\{x \mid px\}$
the	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Me}$	$\lambda p.x$ if $p = \{x\}$ else $\#$
a	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{De}$	$\lambda p.\lambda s.\,\{\langle x, x + s\rangle \mid px\}$
no	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Ce}$	$\lambda p.\lambda c.\neg\exists x.px \wedge c\, x$
every	$(\mathtt{e} \to \mathtt{t}) \to \mathtt{Ce}$	$\lambda p.\lambda c.\forall x, px \Rightarrow cx$

Table 3: λ-calculus representation of the english language $\mathcal{L}$

Constructor	`fmap`	Typeclass
$\mathtt{G}(\tau) = \mathtt{r} \to \tau$	$\mathtt{G}\varphi(x) = \lambda r.\varphi(xr)$	Monad
$\mathtt{W}(\tau) = \tau \times \mathtt{t}$	$\mathtt{W}\varphi((a, p)) = (\varphi a, p)$	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
$\mathtt{M}(\tau) = \tau + \bot$	$\mathtt{M}\varphi(x) = \begin{cases} \varphi(x) & \text{if } \Gamma \vdash x : \tau \\ \# & \text{if } \Gamma \vdash x : \# \end{cases}$	Monad

Table 4: Denotations for the functors used

CN(P)	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t})$	$\Pi(p) = \lambda x.\,(px \wedge \lvert x\rvert \geq 1)$
ADJ(P)	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t})$	$\Pi(p) = \lambda x.\,(px \wedge \lvert x\rvert \geq 1)$
	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to (\mathtt{e} \to \mathtt{t})$	$\Pi(p) = \lambda \nu.\lambda x.\,(p\,(\nu)\,(x) \wedge \lvert x\rvert \geq 1)$
NP	$\Gamma \vdash p : \mathtt{e}$	$\Pi(p) = p$
	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to \mathtt{t}$	$\Pi(p) = \lambda \nu.p\,(\Pi\nu)$
IV(P)/VP	$\Gamma \vdash p : \mathtt{e} \to \mathtt{t}$	$\Pi(p) = \lambda o.\,(po \wedge \lvert x\rvert \geq 1)$
TV(P)	$\Gamma \vdash p : \mathtt{e} \to \mathtt{e} \to \mathtt{t}$	$\Pi(p) = \lambda s.\lambda o.\,(p\,(s)\,(o) \wedge \lvert s\rvert \geq 1)$
REL(P)	$\Gamma \vdash p : \mathtt{e} \to \mathtt{t}$	$\Pi(p) = \lambda x.\,(px \wedge \lvert x\rvert \geq 1)$
DET	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to ((\mathtt{e} \to \mathtt{t}) \to \mathtt{t})$	$\Pi(p) = \lambda \nu.p\,(\Pi\nu)$
	$\Gamma \vdash p : (\mathtt{e} \to \mathtt{t}) \to \mathtt{e}$	$\Pi(p) = \lambda \nu.p\,(\Pi\nu)$

Table 5: Definition for the Π Plural Functor

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

$$\varphi : \underset{\lambda k.\lambda\,(a,g)\,.kag}{\left(\alpha \to \mathtt{G}\beta\right) \to \mathtt{W}\alpha \to \beta} \quad \text{and } \psi : \underset{\lambda c\lambda a\lambda g.c\,(a,g)}{\left(\mathtt{W}\alpha \to \beta\right) \to \alpha \to \mathtt{G}\beta}$$

where $\varphi \circ \psi = \mathrm{id}$ and $\psi \circ \varphi = \mathrm{id}$ on the properly defined sets. Now it is easy to see that φ and ψ do define a natural transformation and thus our adjunction $\mathtt{W} \dashv \mathtt{G}$ is well-defined, and that its unit η is $\psi \circ \mathrm{id}$ and its co-unit ε is $\varphi \circ \mathrm{id}$.

As a reminder, this means our language canonically defines a monad $\mathtt{G} \circ \mathtt{W}$. Moreover, the co-unit of the adjunction provides a canonical way for us to deconstruct entirely the comonad $\mathtt{W} \circ \mathtt{G}$, that is we have a natural transformation from $\mathtt{W} \circ \mathtt{G} \Rightarrow \mathrm{Id}$.

2.4 Modality, Plurality, Aspect, Tense

We will now explain ways to formalise higher-order concepts as described in Section 1.1.3.3.

First, let us consider in this example the plural concept. Here we do not focus on whether our denotation of the plural is semantically correct, but simply on whether our modelisation makes sense. As such, we give ourselves a way to measure if a certain x is plural our not, which we will denote by $\lvert x\rvert \geq 1$. It is important to note that if τ is a type, and $\Gamma \vdash x : \tau$ then we should have $\lvert \Pi x\rvert \geq 1 = \top$[7]. We then need to define the functor Π which models the plural. We do not need to define it on types in any way other than $\Pi\,(\tau) = \Pi\tau$. We only need to look at the transformation on arrows[8]. This one depends on the *syntactic*[9] type of the arrow, as seen in Table 3.

See that our functor only acts on the which return a boolean by adding a plurality condition, and is simply passed down else. Note however there is an issue with the **NP** category: in the case where a **NP** is constructed from a determiner and a common noun (phrase) the plural is not passed down. This comes from the fact that in this case, either the determiner or the common noun (phrase) will be marked, and by functoriality the plural will go up the tree.

Now clearly our functor verifies the identity and composition laws and works as theorised in Section 1.1.3.3. To understand a bit more why this works as described in our natural transformation rules, consider $\Pi\,(\textbf{which})$. The result will be of type $\Pi\left(\mathtt{S}\,(e)\right) = \left(\Pi \circ \mathtt{S}\right)(e)$. However, when looking at our transformation rule (and our functorial rule) we see that the result will actually be of type $\mathtt{S}\,(\Pi e)$ which is exactly the expected type when considering the phrase

[7] This is a typing judgement !

[8] `fmap`, basically.

[9] Actually it depends on the word considered, but since the denotations (when considered effect-less) provided in Table 3 are more or less related to the syntactic category of the word, our approximation suffices.

which p. The natural transformation θ we set is easily inferable:

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

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

Now, in quite a similar way, we can create functors from any sort of judgement that can be seen as a function $\mathtt{e} \to \mathtt{t}$ in our language[10]. Indeed, we simply need to replace the $|p| \geq 1$ in most of the definitions by our function and the rest would stay the same. Remember that our use of natural transformations is only there to allow for possible under-markings of the considered effects and propagating the value resulting from the computation of the high-order effect.

[10] An easy way to define those would be, similarly to the plural, to define a series of judgement from a property that could be infered.

A Other Considered Things

A.1 Typing with a Product Category (and a bit of polymorphism)

Another way to start would be to consider product categories: one for the main type system and one for the effects. Let $\mathcal{C}_0$ be a closed cartesian category representing our main type system. Here we again consider constants and full computations as functions $\bot \to \tau$ or $\tau \in \mathrm{Obj}(\mathcal{C}_0)$. Now, to type functions and functors, we need to consider a second category: We consider $\mathcal{C}_1$ the category representing the free monoid on $\mathcal{F}(\mathcal{L})$. Monads and Applicatives will generate relations in that monoid. To ease notation we will denote *functor types* in $\mathcal{C}_1$ as lists written with head on the left.

Finally, let $\mathcal{C} = \mathcal{C}_0 \times \mathcal{C}_1$ be the product category. This will be our typing category. This means that the real type of objects will be $(\bot \to \tau, [])$, which we will still denote by τ. We will denote by $F_n \cdots F_0 \tau$ the type of an object, as if it were a composition of functions[11].

In that paradigm, functors simply append to the head of the *functor type* (with the same possible restrictions as before, though I do not see what they would be needed for) while functions will take a polymorphic form: $x : L\tau_1 \mapsto \varphi x : L\tau_2$ and φ's type can be written as $\star\tau_1 \to \star\tau_2$.

B Bibliography

Andrej Bauer and Matija Pretnar. 2014. An Effect System for Algebraic Effects and Handlers. *Logical Methods in Computer Science*, Volume 10, Issue 4.

Dylan Bumford. Effectful composition in natural language semantics - Introducing functors.

Dylan Bumford and Simon Charlow. Effect-driven interpretation.

Irene Heim and Angelika Kratzer. 1998. *Semantics in Generative Grammar*. Number 13 in Blackwell Textbooks in Linguistics. Blackwell, Malden, MA.

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

Paul-André Melliès and Noam Zeilberger. 2015. 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. ACM.

E. Moggi. 1989. Computational lambda-calculus and monads. In *[1989] Proceedings. Fourth Annual Symposium on Logic in Computer Science*, pages 14–23, Pacific Grove, CA, USA. IEEE Comput. Soc. Press.

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

Gordon D. Plotkin and Matija Pretnar. 2013. Handling Algebraic Effects. *Logical Methods in Computer Science*, Volume 9, Issue 4.

Birthe van den Berg and Tom Schrijvers. 2024. A framework for higher-order effects & handlers. *Science of Computer Programming*, 234:103086.

Philip Wadler. 1989. 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. Association for Computing Machinery.

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

[11] It is!