A Taste of Categorical Semantics

Marco Paviotti

January 22, 2025

Contents

1 Introduction

When learning an abstract mathematical concept, it is helpful to have a concrete notion of the subject being studied; without this, the abstraction may lack meaningful context. There are probably two approaches to learning category theory in computer science: through the lens of mathematics or through that of functional programming. While the mathematical perspective is arguably the most rigorous, many computer scientists may find the functional programming perspective more accessible. Indeed, some authors have already chosen to adopt this path [6].

In these notes, we take a different approach – one which lies between mathematics and programming languages. This approach aligns with the mathematical treatment of programming languages known as *denotational semantics*. The idealized concept in this context is that a *type* can be viewed as a *set*, and a *program* as a *function* between sets. However, as programming languages incorporate more features, maintaining this simplistic view becomes increasingly challenging. By employing category theory, we generalize this perspective: a type corresponds to an *object*, and a program corresponds to an *arrow*. With this idea in mind, we will model the simply typed λ-calculus, predicative polymorphism and λ-calculi with computational effects.

For the interested reader who wants to dive deeper in these subjects, there is a plethora of very well-written books about category for a more in-depth introduction on the subject [1, 4].

In these notes, we will try to convince the reader that category theory covers a wide variety of subjects; from logic to type theory, hence programming, algebra and topology and it is therefore a subject worth studying for the reader's own good.

To give a taste for why this is true let us look at three most common axioms which we can find in logic, type theory and algebra. In particular, reflexivity and transitivity. Reflexivity states that every proposition A entails itself, writen $A \vdash A$ in logic. In type theory, this corresponds to the *variable rule*, which states that if a variable x has type A, written $x : A$, then we can type check the program $x : A$. Similarly, transitivity ensures that proofs of propositions can be chained: if A entails B and B entails C then A entails C. This is similar to what happens in type theory: if t is a program of type B with an open variable of type A and t' is a program of type C open in $y : B$ then substituting t for y in t' yields a program of type C open in $x : A$. In algebra, a *preorder* has exactly the same properties as for all A in a preorder we have $A \leq A$ and for all A, B, C, if $A \leq B$ and $B \leq C$ then obviously $A \leq C$. The table below summarises the concepts we explained so far:

Logic	Type Theory	Algebra
$A \vdash A$	$x : A \vdash x : A$	$A \leq A$
$\dfrac{A \vdash B \quad B \vdash C}{A \vdash C}$	$\dfrac{x : A \vdash t : B \quad y : B \vdash t' : C}{x : A \vdash \{t/y\}t' : C}$	$\dfrac{A \leq B \quad B \leq C}{A \leq C}$

It should now be evident that three seemingly disconnected areas of mathematics – logic, computer science, and algebra – are deeply interconnected. This connection has been termed *computational trinitarianism*, the idea that logic, type theory, and algebra are three perspectives on the same underlying objects. This perspective reinforces the thesis that computability is an inherent concept in the foundations of mathematics.

In 1945, Saunders MacLane and Samuel Eilenberg, while working on algebraic topology, observed recurring patterns in algebra that could be generalized. He developed an abstract axiomatization encompassing many aspects of algebra, which led to the formulation of the *axioms of a category* [4]. A category consists of objects and morphisms (arrows) between them, governed by two fundamental axioms. The first, *identity*, reflects reflexivity by requiring an identity morphism $\mathrm{id}_A : A \longrightarrow A$ for every object. The second, *composition*, embodies transitivity by allowing morphisms $f : A \longrightarrow B$ and $g : B \longrightarrow C$ to compose into $g \circ f : A \longrightarrow C$.

$$\begin{tikzcd}
A \arrow[loop right, "id_A"] & & A \arrow[r, "f"] \arrow[rr, bend right, "g\circ f"'] & B \arrow[r, "g"] & C
\end{tikzcd}$$

The reader is encouraged to keep these analogies in mind throughout this article, as they provide intuitive insights into the abstract categorical concepts we will explore.

Lastly, since category theory is formulated on top of set theory, we will briefly recap the basic notions of sets, size, families of sets, and Russell's Paradox in the next section. The Russell's Paradox, in particular, is a pivotal result in naive set theory, offering insights into the design choices behind category theory and highlighting certain challenges related to polymorphism in programming languages.

2 Elements of Set Theory

A *set* (or class) is an unordered collection of objects called *elements*. If an object a is an element of a set X, we write $a \in X$ and say "a belongs to X."

One of the most fundamental sets is the *empty set*, denoted $\emptyset$, which contains no elements. It is important to distinguish between $\emptyset$ and the *singleton set* $\{\emptyset\}$, which contains the empty set as its only element.

Other notable sets include the set of *natural numbers*, $\mathbb{N} = \{1, 2, 3, \dots\}$, and the set of *natural numbers with zero*, $\mathbb{N}_0 = \{0, 1, 2, 3, \dots\}$.

Given two sets A and B, we can construct their *Cartesian product*, $A \times B$, which is the set of all ordered pairs (a, b) such that $a \in A$ and $b \in B$. Similarly, the *union* of two sets A and B, written $A \cup B$, is the set containing all elements of A and B, with duplicates removed. A related concept is the *disjoint union*, $A \uplus B$, which pairs elements with tags to distinguish their origin: $(1, a)$ for $a \in A$ and $(2, b)$ for $b \in B$.

2.0.1 Relations and Functions

A *relation* $R \subseteq A \times B$ is a subset of the Cartesian product $A \times B$, linking certain elements of A to elements of B. A *function* is a special type of relation $f \subseteq A \times B$ such that for every $a \in A$, there exists exactly one $b \in B$ related to it. The set of all functions from A to B is denoted $A \longrightarrow B$.

Two special cases of functions are worth noting. First, there is only one function from the empty set $\emptyset$ to any set A, which is the empty relation $! \subseteq \emptyset \times A$. Second, there is only one function from any set A to a singleton set $\{*\}$, which maps every element $a \in A$ to $*$. This function is also denoted !, and its meaning is typically clear from context.

2.0.2 Cardinality and Isomorphism

The *size* or *cardinality* of a set measures how many elements it contains. Two sets are said to have the same cardinality, or to be *isomorphic*, if there exists a bijective function $f : A \longrightarrow B$ with an inverse $f^{-1} : B \longrightarrow A$ such that $f(f^{-1}(x)) = x$ and $f^{-1}(f(x)) = x$ for all x.

A set A is *finite* if it is isomorphic to the set $\{m \in \mathbb{N} \mid m \leq n\}$ for some $n \in \mathbb{N}$. In this case, we can enumerate its elements as $A = \{a_1, a_2, \dots, a_n\}$. A set is *infinite* if it is not finite. A set is *countable* if it is isomorphic to the natural numbers $\mathbb{N}$.

2.0.3 Indexed Families of Sets

For a set I, an *indexed family of sets* is a collection of sets $\{A_i\}_{i \in I}$, where each set A_i is associated with an index $i \in I$. If I is finite, we can write the union and Cartesian product of these sets as:

$$A_1 \cup A_2 \cup \cdots \cup A_n \quad \text{and} \quad A_1 \times A_2 \times \cdots \times A_n.$$

If I is infinite, the *union* of the family is:

$$\bigcup_{i \in I} A_i = \{a \in A_i \mid i \in I\},$$

and the *dependent product* (or *infinite product*) is the set of functions $f : I \to \bigcup_{i \in I} A_i$ such that $f(i) \in A_i$ for all $i \in I$:

$$\Pi_{i \in I} A_i = \{f : I \to \bigcup_{i \in I} A_i \mid f(i) \in A_i\}.$$

2.0.4 The Russel's Paradox

Russell's Paradox is a fundamental problem in set theory, discovered by Bertrand Russell in 1901. It reveals a contradiction in naive set theory, which allowed the formation of any set based on a defining property, without restrictions. The paradox shows that such a theory can lead to logical inconsistencies.

The paradox arises when we consider the set of all sets that do not contain themselves as a member. Let's define this set as R. Formally, R is the set of all sets that do not contain themselves as a member. In other words:

$$R = \{x \mid x \notin x\}$$

Here, R is the set of all sets x such that x does *not* contain itself as a member.

Now, the central question of the paradox is: **Does the set R contain itself?**

To answer this, we explore two possibilities. The case when $R \in R$ and the case when $R \notin R$. If R is a member of itself, then by the definition of R, it must not contain itself (because R is the set of all sets that do not contain themselves). Therefore, if $R \in R$, it must follow that $R \notin R$, which is a contradiction. If R is *not* a member of itself, then by the definition of R, it must contain itself (because R is the set of all sets that do not contain themselves, and R would be one of those sets). Therefore, if $R \notin R$, it must follow that $R \in R$, which is also a contradiction.

This contradiction shows that the assumption that such a set R can exist leads to an inconsistency. The paradox demonstrates that naive set theory, which allowed for the creation of sets like R, is inherently flawed.

2.0.5 The Axiom of Regularity (Foundation)

The Axiom of Regularity, also known as the Axiom of Foundation, is one of the axioms in Zermelo-Fraenkel set theory (ZF), designed to prevent certain paradoxes like Russell's Paradox.

The Axiom of Regularity states:

$$\forall A\, (A \neq \emptyset \Rightarrow \exists x \in A\, (x \cap A = \emptyset))$$

This means that for every non-empty set A, there exists an element $x \in A$ such that x and A are disjoint sets.

The Axiom of Regularity essentially says that *no set can be a member of itself* (directly or indirectly), and it prevents sets from containing themselves or forming cycles. In other words, it ensures that sets are well-founded, meaning they cannot "loop back" on themselves in any way.

In the case of Russell's Paradox, we defined a set R as:

$$R = \{x \mid x \notin x\}$$

The paradox arose because the set R seemed to both contain itself and not contain itself, depending on the assumption. The Axiom of Regularity helps avoid such contradictions by ensuring that no set can be a member of itself. It guarantees that sets like R, which would allow self-referencing and circular definitions, cannot exist.

Let A be a set, and apply the axiom of regularity to the singleton set containing A, that is $\{A\}$. By the axiom of regularity there must be an element of A which is disjoint from $\{A\}$. Since A is the only element of $\{A\}$, it must be that A is disjoint from $\{A\}$. Therefore, since $A \cap \{A\} = \emptyset$ that means that A does not contain itself by definition of intersection.

3 Categories

As explained above, the intuition the reader should have is that when talking about well-typed programming languages, types should be regarded as *objects* and programs should be regarded as *arrows*. This sort of motivates the definition of a category:

Definition 3.1 (Category)**.** *A category C is a collection of objects $A, B, C \dots$ denoted by Obj(C) and a set of arrows $f, g, h \dots$ denoted by Arr(C). Additionally, for each object A there exists an identity arrow $id_A : A \to A$ such that and for arrows $f : A \to B$ and $g : B \to C$ we there exists an arrow $g \circ f : A \to C$. We picture these as mentioned in the introduction:*

$$\begin{tikzcd} A \arrow[loop right, "id_A"] & A \arrow[r, "f"] \arrow[rr, bend right, "g\circ f"'] & B \arrow[r, "g"] & C \end{tikzcd}$$

Additionally, these arrows obey the identity and associativity laws:

$$id_A \circ f = f \circ id_A = f$$
$$f \circ (g \circ h) = (f \circ g) \circ h$$

Examples of categories are the category of sets, denoted by **Set**, where objects are sets and arrows are functions, the category of sets and relations **Rel**, the category of partial order sets **Pos**, the category of groups **Grp** of groups and group homomorphisms, the category of topological spaces **Top** of topological spaces and continuous functions between them.

To avoid clutter, we simply write $X \in C$ for an object in a category C and $f \in C$ for a morphism in a category C. For objects $X, Y \in C$ we write $C(X, Y)$ for the collection of morphisms from X to Y which we call the homset.

The fact that a category is defined in terms of collections objects and morphisms is to avoid paradoxes such as the one in Section 2.0.4. For example, the category **Set** is too big for the objects to form a set, since the set of sets does not exists. When the collection of objects is a set we say the category is *small*. Similarly when the homset is a set we say the category is *locally small*.

3.1 Initial and terminal objects

In logic a false proposition entails any formula A. In category theory this is expressed in terms of initial objects. The initial object is an object denoted by 0 such that for every other object A there exist a unique arrow between 0 and A. We draw a dashed arrow as follows to indicate the arrow is unique:

$$\begin{tikzcd} 0 \arrow[r, dotted, "!_A"] & A \end{tikzcd}$$

In other words, there is a unique proof which takes a proof of the false statement and returns a proof of any statement A. This can also be interpreted in programming as the program from the empty type into any type. Intuitively, the only way to produce something of type A is to case analyse on the empty type, but because this type does not contain anything the program has nothing further to compute and the case is vacuous.

Similarly, the *terminal* object 1 represent the true statement or the type with only one element in it, the *unit type*. In programming terms, given any input of type A there exists exactly one program producing an element of the unit type, that is the program which discards the input and returns the single element in the unit type. Categorically, and it is such that for every object A there is a unique arrow into the terminal object:

$$\begin{tikzcd} A \arrow[r, dotted, "!_A"] & 1 \end{tikzcd}$$

It is important to know that such objects in category theory are unique only up-to isomorphism.

In **Set**, the empty set $\emptyset$ is the initial object since there is a unique function from the empty set to any other set A, that is the empty function or empty relation. Similarly, the terminal object is any set containing exactly one element, the singleton set. Consider the set $\{*\}$. For any other given set A there is a unique function into $\{*\}$ which is the constant function mapping every element $x \in A$ into $\{*\}$. Notice that there are many terminal objects in **Set**, but they are all isomorphic in that they all contain only one element. Also **Set** is a special category of sorts, in fact, it enjoys the property that the set of morphisms from 1 to any set A is isomorphic to A itself since any function $1 \xrightarrow{x} A$ can map into exactly one element in A.

Exercise 3.1. *Prove that for all sets $X \in$ **Set**, the homset **Set**$(1, X)$ is in bijective correspondence with X.*

3.2 The Natural Numbers object

The *natural numbers object* (NNO) is an abstract representation of the natural numbers $\mathbb{N}$ within a category. It consists of an object $\mathbb{N}$ along with two morphisms: the zero morphism $1 \xrightarrow{z} \mathbb{N}$ and the successor morphism $\mathbb{N} \xrightarrow{s} \mathbb{N}$ These morphisms encode the structure of the natural numbers, with $z$corresponding to the base element 0 s representing the successor function, which maps a number $n \mapsto n + 1$.

The NNO is characterized by a universal property: for any object X in the category and any pair of morphisms $1 \xrightarrow{f} X$ (representing a base case) and $X \xrightarrow{g} X$ (representing a recursive step), there exists a unique morphism $\mathbb{N} \xrightarrow{h} \mathbb{N}$ such that $h \circ z = f$ and $h \circ s = g \circ h$

$$\begin{tikzcd}
1 \arrow[r, "z"] \arrow[dr, "f"'] & \mathbb{N} \arrow[r, "s"] \arrow[d, dotted, "h"] & \mathbb{N} \arrow[d, dotted, "h"] \\
& X \arrow[r, "g"] & X
\end{tikzcd}$$

This abstract construction provides a general way to represent natural numbers and their properties in various categories, such as the category of sets or more complex structures like toposes.

3.3 Isomorphisms

Isomorphism is a fundamental concept that captures the idea of two objects being "essentially the same". An isomorphism between two objects indicates that they are structurally identical, even if they might appear different externally. While the concept of "essentially the same" is central to isomorphism, it is not always straightforward to understand what this means in different settings. An isomorphism in **Set** is a pair of functions which are inverses to each other. This corresponds to saying that two sets are isomorphism if they have the same cardinality, which is equivalent to saying that there exists a function $f : A \longrightarrow B$ such that is surjective and injective.

However, consider the the category **Pos** of partial order sets and order preserving functions. The following are two posets which are not isomorphic:

$$\begin{tikzcd}
1 & & 2 & 1 & & 2 \\
& \perp \arrow[ul] \arrow[ur] & & & \perp \arrow[ur] &
\end{tikzcd}$$

since in the right-hand side poset the $\perp$ element is not ordered with 1. Despite the fact that these two posets are in bijection they are not isomorphic because any bijection would be able to preserve the order $\perp \leq 1$ from the left to the right-hand side poset (while still being a bijection).

Category theory abstracts the notion of isomorphism by stating that two objects are isomorphic if and only if there exists a pair of morphisms which are inverse to each other

Definition 3.2 (Isomorphism)**.** *Two given objects A and B are isomorphic, written $A \cong B$ iff there exists an arrow $f : A \longrightarrow B$ that has an inverse $g : B \longrightarrow A$ such that*

$$g \circ f = id_A \qquad f \circ g = id_B$$

This definition depends of course on the definition of morphism, in particular, an isomorphism is defined by what the arrows look like and by what we can observe through them rather than what are the objects themselves.

Exercise 3.2. *Let X and Y be two initial objects. Prove that $X \cong Y$. Conclude that initial objects are unique up-to isomorphism.*

Exercise 3.3. *Let* $0 \xrightarrow{!_A} A$ *be the unique morphism from the initial object into an object* A *and* $A \xrightarrow{f} B$ *be a morphism. Prove that* $f \circ !_A$ *is equal to* $!_B$.

3.4 Opposite categories

An *opposite category* refers to a way of reversing the structure of a given category. If we have a category C, its opposite category, denoted C^{op}, is formed by reversing the direction of all morphisms while keeping the same objects.

Definition 3.3 (Opposite category)**.** *For a category* C*, there is a category* C^{op} *which has as objects the same objects as* C *and for every morphism* $f : A \to B$ *in* C *a morphism* $f^{\mathsf{op}} : B \to A$

$$f \in C(X,Y) \Longleftrightarrow f^{\mathsf{op}} \in C^{\mathsf{op}}(Y,X)$$

In other words, for every morphism $f : A \to B$ in C, there is a corresponding morphism $f^{\mathrm{op}} : B \to A$ in C^{op}. The composition of morphisms in C^{op} is defined in reverse order compared to C, meaning that if $f : A \to B$ and $g : B \to C$ are morphisms in C, their composition in C^{op} is given by $g^{\mathrm{op}} \circ f^{\mathrm{op}}$, corresponding to $(f \circ g)^{\mathrm{op}}$ in C.

The opposite category is a useful tool for exploring dualities, where a statement about C has a corresponding dual statement about C^{op}. We will go back to this topic when we introduce the concept of a functor in Section 5.

4 Products, CoProducts and Exponentials

In this section we are going to take a look at the categorical semantics of the simply typed λ-calculus. This is a λ-calculus with natural numbers, pairs and function types therefore it is only natural that to model these we need their logical and algebraic counterparts.

4.1 Products

In logic, the *and* operator is introduced by stating that if Γ is a set of true propositions which entails A and this context Γ entails also B then of course $\Gamma \vdash B$. This is called the introduction rule of the product:

$$\frac{\Gamma \vdash A \qquad \Gamma \vdash B}{\Gamma \vdash A \wedge B}$$

The product has also two elimination rules given by:

$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\;\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

In algebra, a set which possess this structures is called a *lower semilattice*. That is a set X such that for every $a, b \in X$, there exists an element $a \wedge b$ called the *meet* or the *greatest lower bound* of a and b which has the property that for every other lower bound z, that is $z \leq a$ and $z \leq b$ we have that $z \leq a \wedge b$.

$$\begin{tikzcd}
x \arrow[rrd, dash, "\leq"] \arrow[rrdd, dash, "\leq"'] & & & & y \arrow[lld, dash, "\leq"'] \arrow[lldd, dash, "\leq"] \\
 & & x \wedge y \arrow[d, dash, "\leq"] & & \\
 & & z & &
\end{tikzcd}$$

We proceed now to generalise the concept of greatest lower bound to the categorical notion of product $A \times B$.

Definition 4.1 (Product). *For two objects A and B the* product *is an object P equipped with two arrows $\pi_1 : P \longrightarrow A$ and $\pi_2 : P \longrightarrow B$ such that for any object Z with arrows $f : Z \longrightarrow A$ and $g : Z \longrightarrow B$ there exists a unique morphism $h : Z \longrightarrow P$ making the following diagram commute:*

$$\begin{tikzcd}
A & P \arrow[l, "\pi_1"'] \arrow[r, "\pi_2"] & B \\
 & Z \arrow[ul, "f"] \arrow[u, dotted, "h"'] \arrow[ur, "g"'] &
\end{tikzcd}$$

We write $A \times B$ for the product of two objects A and B and $\langle f, g \rangle$ for h.

Note that this definition corresponds to the notion of product in **Set**

$$A \times B \overset{\Delta}{=} \{\langle a, b \rangle \mid a \in A \text{ and } b \in B\}$$

where the projections π_1 and π_2 are simply the functions discarding one component of the pair $\pi_1(a, b) =$ and $\pi_2(a, b) = b$ and for every element $1 \xrightarrow{a} A$ and $1 \xrightarrow{b} B$ the pairing is defined by $(a, b) \mapsto \langle a, b \rangle$ Note that the existence condition of the pairing function enforces that every element of A and B are in the product (no noise) and the uniqueness condition ensures that there is are no more elements in $A \times B$ than the ones that are coming from A and B (no junk).

Notice also that we could have defined the product in many other ways, but as long as the categorical definition is satisfied, all these definitions are isomorphic. The reader should convince themselves that is true by proving the following proposition:

Proposition 4.1 (Products are unique up to isomorphism). *Let C be a category with products. Then for all objects A and B, if P and Q are both products of A and B then $P \cong Q$.*

Another isomorphic definition of product in **Set** could be the following one:

$$A \times B = \{(b, a) \mid a \in A \text{ and } b \in B\}$$

or this one

$$A \times B = \{(b, a, a) \mid a \in A \text{ and } b \in B\}$$

Exercise 4.1. *Prove Proposition 4.1. Hint: you have to prove that for two objects A and B if you have two products P and P' for the same two objects you can derive an isomorphism. This is constructed by using the universality property of products.*

4.2 Coproducts

In logic, the *or* operator is introduced by stating that if Γ is a set of true propositions that entails A, or if Γ entails B, then $\Gamma \vdash A \vee B$. This is called the introduction rule of the coproduct:

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}$$

The coproduct has one elimination rule given by:

$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

In algebra, a set with this structure is called an *upper semilattice*. That is, a set X such that for every $a, b \in X$, there exists an element $a \vee b$, called the *join* or the *least upper bound* of a and b, with the property that for every other upper bound z (i.e., $z \geq a$ and $z \geq b$), we have $z \geq a \vee b$. This can be visualized as follows:

$$\begin{tikzcd}
& z \arrow[d, dash, dashed, "\geq"] \arrow[ddl, dash, "\geq"'] \arrow[ddr, dash, "\geq"] & \\
& x \vee y \arrow[dl, dash, "\geq"'] \arrow[dr, dash, "\geq"] & \\
x & & y
\end{tikzcd}$$

We now generalize the concept of least upper bound to the categorical notion of a coproduct $A + B$.

Definition 4.2 (Coproduct)**.** *For two objects A and B, the* coproduct *is an object C equipped with two arrows $i_1 : A \longrightarrow C$ and $i_2 : B \longrightarrow C$ called* injections *such that for any object Z with arrows $f : A \longrightarrow Z$ and $g : B \longrightarrow Z$, there exists a unique morphism $h : C \longrightarrow Z$ making the following diagram commute:*

$$\begin{tikzcd}
A \arrow[r, "i_1"] \arrow[dr, "f"'] & C \arrow[d, dotted, "h"] & B \arrow[l, "i_2"'] \arrow[dl, "g"] \\
& Z &
\end{tikzcd}$$

We write $A + B$ for the coproduct of two objects A and B and $[f, g]$ for h.

This definition corresponds to the notion of a disjoint union in **Set**:

$$A + B \triangleq \{(a, 1) \mid a \in A\} \cup \{(b, 2) \mid b \in B\},$$

where the injections i_1 and i_2 are defined as:

$$i_1(a) = (a, 1) \quad \text{and} \quad i_2(b) = (b, 2).$$

For any element $f : A \longrightarrow Z$ and $g : B \longrightarrow Z$, the unique morphism $[f, g]$ is defined by:

$$[f,g](x) = \begin{cases} f(a) & \text{if } x = (a,1), \\ g(b) & \text{if } x = (b,2). \end{cases}$$

The existence condition ensures that every element of A and B is included in the coproduct (no omission), while the uniqueness condition ensures that there are no additional elements in $A + B$ (no duplication).

Notice that we could define the coproduct in many ways, but as long as the categorical definition is satisfied, all these definitions are isomorphic. The reader should verify this by proving the following proposition:

Proposition 4.2 (Coproducts are unique up to isomorphism)**.** *Let C be a category with coproducts. Then for all objects A and B, if C and D are both coproducts of A and B, then $C \cong D$.*

Exercise 4.2. *Prove Proposition 4.2. Hint: you need to show that for two objects A and B, if C and C' are both coproducts, you can construct an isomorphism using the universal property of coproducts.*

4.3 Exponentials

In logic, the introduction rule of the implication is typically written as follows:

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \subset B}$$

where $\subset$ is how traditionally implication has been written and is reminiscent of the fact that the information A provides is a subset of the information that B provides. The introduction rule means that if we can derive B from a context Γ and a proof of A then we can derive that given Γ we can prove A implies B. The elimination rule of the implication is called "modus ponens" rule is typically written as follows:

$$\frac{\Gamma \vdash A \subset B \qquad \Gamma \vdash A}{\Gamma \vdash B}$$

meaning that if Γ entails A implies B and we also have that Γ entails A then we can derive B.

The algebraic structure modelling this operator is called an *Heyting algebra*. An Heyting algebra consists of a set X with finite meets ($\wedge$) and exponents ($\Rightarrow$) such that the following universal property holds

$$z \wedge x \leq y \iff z \leq x \Rightarrow y$$

for all x, y, z.

This time the categorical definition will not match exactly the abstract definition of the Heyting algebra. However we can show we can reduce the following definition to the one above for preorders.

Definition 4.3 (Exponentials)**.** *An* exponential *represents the* internal *function type of a language. The triangular diagram in (1) defines the universality property of an exponential. This is an object which we denote by B^A, for $A, B \in C$ such that there exists an* evaluation map $\epsilon : B^A \times A \longrightarrow B$ *which intuitively applies the input of type A to a function which takes A to B and such that for every map $Z \times A \xrightarrow{f} B$ there exists a unique map $Z \xrightarrow{\Lambda f} B^A$ such that the diagram in (1) commutes.*

$$\begin{tikzcd}
B^A & & B^A \times A \arrow[r, "\epsilon"] & B \\
Z \arrow[u, dotted, "\Lambda f"] & & Z \times A \arrow[u, "\Lambda f \times"] \arrow[ur, "f"'] &
\end{tikzcd} \tag{1}$$

Notice, here Λ can be seen as the currying operation taking maps of type $Z \times A \longrightarrow B$ to maps to type $Z \longrightarrow B^A$. Conversely, if we happen to have a map $Z \xrightarrow{g} B^A$ then we can easily construct a map $Z \times A \xrightarrow{\vee g} B$ by $\epsilon \circ \langle g \circ \pi_1, \pi_2 \rangle$. Thus it can be proven that there is a correspondence of arrows, in other words, a correspondence of homsets

$$\Lambda : C(Z \times A, B) \cong C(Z, B^A) : \vee$$

which corresponds to our definition of Heyting exponential and it is the categorical definition of currying and uncurrying.

4.4 Semantics of the Simply Typed λ-Calculus

In this section we look at a categorical semantics for the Simply-Typed λ-calculus (STLC) which was introduced by Alonzo Church in 1940. Originally, the calculus only considered function types, but here we add the unit type, the natural numbers object, finite product and coproducts so that we can show off the category theory we introduced so far.

4.4.1 Syntax

We first define the syntax of STLC by defining the set of types, the set of terms and the typing judgment relation. The set of types Types also inductively as follows

$$\begin{array}{rcll}
A, B \in \text{Types} & ::= & \mathsf{unit} & \text{(unit type)} \\
& | & \mathsf{nat} & \text{(natural numbers)} \\
& | & A \times B & \text{(products)} \\
& | & A + B & \text{(coproducts)} \\
& | & A \longrightarrow B & \text{(functions)}
\end{array}$$

while the set of λ-terms Terms is inductively defined as follows

$$\begin{array}{rcll}
t \in \text{Terms} & ::= & x & \text{(terms variables)} \\
& | & \underline{n} & \text{(natural numbers)} \\
& | & (t_1, t_2) \mid \text{prj}_1(t) \mid \text{prj}_2(t) & \text{(products)} \\
& | & \text{case } t \text{ of } \{\text{inl}(x) \Rightarrow t_1; \text{inr}(y) \Rightarrow t_2\} \mid \text{inl}(t) \mid \text{inr}(t) & \text{(coproducts)} \\
& | & \lambda x.t \mid t_1 t_2 & \text{(functions)}
\end{array}$$

Equational Laws

Typing Judgment The typing judgement relation $\vdash$ is defined inductively as follows

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash \underline{n} : \mathsf{nat}}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathrm{prj}_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathrm{prj}_2(t) : B} \qquad \frac{\Gamma \vdash t_1 : A \quad \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B}$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathrm{inl}(t) : A + B} \qquad \frac{\Gamma \vdash t : B}{\Gamma \vdash \mathrm{inr}(t) : A + B}$$

$$\frac{\Gamma \vdash t : A + B \quad \Gamma, A \vdash t_1 : C \quad \Gamma, B \vdash t_1 : C}{\Gamma \vdash \mathrm{case}\ t\ \mathrm{of}\ \{\mathrm{inl}(x) \Rightarrow t_1; \mathrm{inr}(y) \Rightarrow t_2\} : C}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

4.4.2 Semantics

The semantics of a programming language is given by the *semantic function* mapping the syntax into the *meaning* of the language. This function is traditionally denoted by $[\![\cdot]\!]$ and pronounced the *semantic brackets*. We use the semantic brackets for both the types, terms and the context interpretation. First we need to interpret the contexts. Since these are essentially finite lists we need a category with *finite products*. Thus define $[\![\Gamma]\!]$ by assuming that every context Γ is a finite list of typed variables $x_1 : A_1, \cdots, x_n : A_n$:

$$[\![x_1 : A_1, \cdots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

Or equivalently, by interpreting the context by induction on the list:

$$\begin{aligned} [\![.]\!] &= 1 \\ [\![\Gamma, x : A]\!] &= [\![\Gamma]\!] \times [\![A]\!] \end{aligned}$$

which is a more intuitive definition for readers familiar with functional programming.

The interpretation of types is then a function $[\![\,]\!] : \text{Types} \to \text{Obj}(C)$ from syntactic types into object of a category enforcing the intuition that in categorical semantics we think of types as objects:

$$\begin{aligned} [\![\mathsf{unit}]\!] &= 1 \\ [\![\mathsf{nat}]\!] &= \mathbb{N} \\ [\![A \times B]\!] &= [\![A]\!] \times [\![B]\!] \\ [\![A \to B]\!] &= [\![B]\!]^{[\![A]\!]} \end{aligned}$$

And, finally, the function $[\![\cdot]\!] : \text{Terms} \to \text{Arr}(C)$ interprets a term as a morphism between two objects in the category C. However, to be more precise we only interpret well-typed terms, thus it would be more correct to say that a typing derivation $\Gamma \vdash t : A$

is interpreted as an arrow $[\![t]\!]$ of type $[\![\Gamma]\!] \to [\![A]\!]$, therefore abusing some notation the correct type of the interpretation would be something like:

$$[\![\Gamma \vdash t : A]\!] : [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![A]\!]$$

For example, if $\Gamma, x : A \vdash t : B$ then the interpretation of t has type

$$[\![\Gamma, x : A \vdash t : B]\!] : [\![\Gamma]\!] \times [\![A]\!] \to [\![B]\!]$$

We now define this function by induction on the typing judgement:

$$\begin{aligned}
[\![\Gamma \vdash () : \mathsf{unit}]\!] &= ! \\
[\![\Gamma \vdash \underline{n} : \mathsf{nat}]\!] &= n \\
[\![\Gamma \vdash x : A]\!] &= \pi_x \\
[\![\Gamma \vdash \mathrm{prj}_1(t) : A]\!] &= \pi_1 \circ [\![t]\!] \\
[\![\Gamma \vdash \mathrm{prj}_2(t) : B]\!] &= \pi_2 \circ [\![t]\!] \\
[\![\Gamma \vdash (t_1, t_2) : A \times B]\!] &= \langle [\![t_1]\!], [\![t_2]\!] \rangle \\
[\![\Gamma \vdash \mathrm{inl}(t) : A]\!] &= i_1 \circ [\![t]\!] \\
[\![\Gamma \vdash \mathrm{inr}(t) : A]\!] &= i_2 \circ [\![t]\!] \\
[\![\Gamma \vdash \mathrm{case}\ t\ \mathrm{of}\ \{\mathrm{inl}(x) \Rightarrow t_1; \mathrm{inr}(y) \Rightarrow t_2\} : A]\!] &= [[\![t_1]\!], [\![t_2]\!]] \circ [\![t]\!] \\
[\![\Gamma \vdash \lambda x.M : A \to B]\!] &= \Lambda [\![M]\!] \\
[\![\Gamma \vdash MN : B]\!] &= \epsilon \circ \langle [\![M]\!], [\![N]\!] \rangle
\end{aligned}$$

where $! : [\![\Gamma]\!] \to 1$ is the unique map into the terminal object and $n : \Gamma \to \mathbb{N}$ is the map disregarding the context and returning the number denoted syntactically by $\underline{n}$. In **Set**, for example, there would be a constant map $\lambda y.n$ for each n.

4.5 Exercises

The following two technical lemmas are needed to solve some of the exercises.

Lemma 4.1 (Substitution)**.** *Suppose* $\Gamma, x : A \vdash t : B$ *and* $\Gamma \vdash u : A$ *are valid typing judgments. Then so is* $\Gamma \vdash t[u/x] : B$ *and the interpretation of* $\Gamma \vdash t[u/x] : B$ *is the composite*

$$[\![\Gamma]\!] \xrightarrow{\langle id, [\![u]\!] \rangle} [\![\Gamma]\!] \times [\![A]\!] \xrightarrow{[\![t]\!]} [\![B]\!]$$

Lemma 4.2 (Weakening)**.** *If* $\Gamma \vdash s : C$ *is a valid typing judgment, so is* $\Gamma, x : A \vdash s : C$ *and the following diagram is commutative*

$$\begin{tikzcd}
{[\![\Gamma]\!] \times [\![A]\!]} \arrow[r, "{[\![\Gamma, x:A \vdash s:C]\!]}"] \arrow[d, "\pi_1"'] & {[\![C]\!]} \\
{[\![\Gamma]\!]} \arrow[ur, "{[\![\Gamma \vdash s:C]\!]}"'] &
\end{tikzcd}$$

Exercise 4.3. *Let C be a cartesian closed category with an initial object 0. Show that for any object X in C, $X \times 0$ is initial. Conclude $X \cong 0$.*

Exercise 4.4. *In this exercise you must construct the central part of the soundness proof for the interpretation of the simply typed λ-calculus in a cartesian closed category.*

Let C be a cartesian closed category. Show that the interpretation is sound with respect to β and η rules for function types, i.e., show that if

$$\Gamma, x : A \vdash t : B$$
$$\Gamma \vdash u : A$$
$$\Gamma \vdash s : A \to B$$

then the following equalities hold

$$[\![(\lambda x.t)u]\!] = [\![t[u/x]]\!]$$
$$[\![\lambda x.(sx)]\!] = [\![s]\!]$$

Here $t[u/x]$ means capture-avoiding substitution *of u for x in t. You do not have to worry about what capture avoiding substitution is, rather you should use Lemma 4.1 and Lemma 4.2, which you do not have to prove. (They can be proved by induction on t and s respectively.)*

5 Functors and Natural Transformations

Definition 5.1 (Functor)**.** *Let C and $\mathcal{D}$ be two categories. A* functor *$F : C \to \mathcal{D}$ is a mapping between categories that associates objects in C to an object $FC \in Obj(C)$ and that has additionally a* functorial action *associating arrows $f \in \hom(A, B)$ to arrows $F(f) \in \hom(FA, FB)$ which additionally preserves identity and composition:*

$$F(id_A) = id_{FA} \qquad F(g \circ f) = F(g) \circ F(f)$$

Every functor preserves isomorphisms:

$$A \cong B \Rightarrow FA \cong FB$$

A functor whose functorial action is surjective is called *full*. whilst when the functorial action is injective the functor is called *faithful*.

Proposition 5.1. *A fully faithful functor F preserves and reflects isomorphisms:*

$$A \cong B \iff FA \cong FB$$

Exercise 5.1. *Prove Proposition 5.1*

Definition 5.2 (Natural Transformation)**.** *For two functors $F, G : \mathcal{C} \to \mathcal{D}$, a* natural transformation *is a family of arrows $\phi_A : FA \to GA$ such that for every arrow $f : A \to B$ the following diagram commutes:*

$$\begin{tikzcd}
FA \arrow[r, "\phi_A"] \arrow[d, "F(f)"'] & GA \arrow[d, "G(f)"] \\
FB \arrow[r, "\phi_B"] & GB
\end{tikzcd}$$

5.1 Categories of Functors

5.2 Categories of Categories

Example 5.1 (Is **Set** $\cong$ **Set**$^{\mathsf{op}}$?)**.** *Whenever we have an isomorphism $f : X \to Y$, every property that holds for X should hold for Y and viceversa.*

Now assume we have an isomorphism (of categories) F : **Set** $\to$ **Set**$^{\mathsf{op}}$ *and consider a map $f : A \to \emptyset$ in* **Set**. *(You should read about initial and terminal objects to understand what $\emptyset$ is).*

Since $\emptyset$ is initial in **Set** *then $F\emptyset$ should be terminal in* **Set**$^{\mathsf{op}}$. *Now $Ff : FA \to F\emptyset$ is a map in* **Set**$^{\mathsf{op}}$ *and because $F\emptyset$ is terminal we have many more arrows $FA \to F\emptyset$ than $A \to \emptyset$. Exercise: Work this out!.*

Example 5.2 (Is **Rel** $\cong$ **Rel**$^{\mathsf{op}}$?)**.** *First off,* **Rel** *has as objects sets X, Y, Z and an arrow $f : X \to Y$ is a relation $f \subseteq X \times Y$. Define now two functors F :* **Rel** $\to$ **Rel**$^{\mathsf{op}}$ *and G :* **Rel**$^{\mathsf{op}}$ $\to$ **Rel** *such that they are one the inverse of the other. We define $FX = X$ and $F(R : X \to Y)(y \in Y) = \{x \mid x\ R\ y\}$ and $GX = X$ and $G(R : Y \to X)(y \in Y) = \{x \mid y\ R\ x\}$*

Verify that this is an isomorphism. (You first need to define the composition of two morphisms (relations) in **Rel***)*

5.3 The Homset Functor

Given a (locally small) category $\mathcal{C}$[1] the homset $\hom_{\mathcal{C}}(A, B)$, for any two objects A, B, induces two functors.

The first is the covariant functor $\hom_{\mathcal{C}}(A, -) : \mathcal{C} \to$ **Set** which sends an object B to the set of arrows between A and B and an arrow $B \to B'$ to the functorial action $\hom_{\mathcal{C}}(A, f) : \hom_{\mathcal{C}}(A, B) \to \hom_{\mathcal{C}}(A, B')$ which sends an arrow $g : A \to B$ to the post-composition $f \circ g : A \to B'$.

The second is the contravariant functor $\hom_{\mathcal{C}}(-, B) : \mathcal{C}^{\mathsf{op}} \to$ **Set** which sends an object A to the set of arrows between A and B. This functor is contravariant because it takes an arrow $A \to A'$ to the functorial action $\hom_{\mathcal{C}}(f, B) : \hom_{\mathcal{C}}(A', B) \to \hom_{\mathcal{C}}(A, B)$ which sends an arrow $g : A' \to B$ to the pre-composition $g \circ f : A \to B$.

[1] A category is small if the collection of the objects forms a set and it is "locally small" if for any two objects A, B the collection of arrows between them is a set, however, we do not delve into size issues in these notes.

5.3.1 Natural Isomorphisms

A natural isomorphism is an *isomorphism of functors*. We the tools provided by category theory this should not be a hard notion to derive since we can form the category of functors $C^{\mathcal{D}}$ and natural transformations betwen them. At this point an isomorphism in this category is precisely an isomorphism of functors.

More precisely, for two functors $F, G : C \to \mathcal{D}$ we say these two functors are isomorphism if there exist a natural transformation $\phi : F \stackrel{\cdot}{\to} G$ which has an inverse $\phi^{-1} : G \stackrel{\cdot}{\to} F$ which is also a natural transformation.

Example 5.3. *For example, the stream functor Str* $\cdot$: **Set** $\to$ **Set** *defined by the greatest solution to the following equation*

$$Str\ A \cong A \times Str\ A \tag{2}$$

is isomorphic to the functor $\hom_{\mathbf{Set}}(\mathbb{N}, -)$*, i.e. the following isomorphism is natural in* A

$$Str\ A \cong \hom_{\mathbf{Set}}(\mathbb{N}, A) \tag{3}$$

Exercise 5.2. *Prove this fact by defining a natural transformation* $\phi_A : Str\ A \to \hom_{\mathbf{Set}}(\mathbb{N}, A)$ *and its inverse.*

In general, any covariant functor F that is isomorphic to the hom functor $\hom_C(A, -)$ for some $A \in \mathrm{Obj}(()C)$ is called *representable*.

Exercise 5.3. *Show that the type of lists as defined below*

$$ListA \cong 1 + A \times ListA \tag{4}$$

is not representable.

Exercise 5.4. *Let C be a locally small category, and let A, B, P be object of C. Show that P is a product of A, B, i.e. there is a product diagram*

$$\begin{tikzcd} A & P \arrow[l] \arrow[r] & B \end{tikzcd}$$

if and only if there is an isomorphism

$$C(X, P) \cong C(X, A) \times C(X, B)$$

that is natural in X, in other words, iff the functors

$$C(-, P) : C^{\mathsf{op}} \to \mathbf{Set}$$
$$C(-, A) \times C(-, B) : C^{\mathsf{op}} \to \mathbf{Set}$$

are (naturally) isomorphic.

6 Limits and Colimits

We are going to proceed up the ladder of abstraction and generalise the notion of arbitrary products and coproducts.

Given a set I and a family of sets indexed by I, $\{A_i\}$ we can form the *dependent product* of these sets $\Pi_{i \in I} A_i$ which informally is the product of all sets A_i

$$A_1 \times A_2 \times A_3 \times \cdots \times A_n \times \ldots$$

Dually we can form the *dependent sum* is denoted by $\sum_{i \in I} A_i$ and can be written informally as follows:

$$A_1 + A_2 + A_3 + \cdots + A_n + \ldots$$

Limits and Colimits are just generalisation of these concepts.

Definition 6.1 (Cone)**.** *For a diagram $D : \mathcal{I} \longrightarrow \mathcal{D}$, a* cone *is an object C such that ther exists a family of maps $\pi_i : C \longrightarrow D_i$ and such that*

$$D(f) \circ \pi_i = \pi_j \tag{5}$$

for every $f : i \longrightarrow j$.

Definition 6.2 (Limit)**.** *For a diagram $D : \mathcal{I} \longrightarrow C$, a* limit *is a universal cone P, that is a cone that for any other cone for D, say C, there exists a unique arrow $h : C \longrightarrow \lim_{i \in \mathcal{I}} D_i$ such that*

$$\pi_i \circ h = f_i \tag{6}$$

for every $i \in \mathcal{I}$. The picture below summarises the situation:

$$\begin{tikzcd}
D_j \arrow[rr, "D(f)"] & & D_{j'} \\
& \varprojlim D_i \arrow[ul, "\pi_j"'] \arrow[ur, "\pi_{j'}"] & \\
& C \arrow[u, dashed, "h"'] \arrow[uul, bend left, "f_j"] \arrow[uur, bend right, "f_{j'}"'] &
\end{tikzcd}$$

We denote the limit of a diagram D by $\varprojlim_{i \in \mathcal{I}} D_i$

Exercise 6.1. *The notion of colimit is obtained by reversing the arrows. Derive it by exercise.*

Dependent Products and Sums The dependent product and dependent sum are just limits and colimits respectively where the category $\mathcal{I}$ is discrete[1]. In this particular case the coherence conditions (5) on the projections are vacuous because there are no meaningful arrows in the index category. This means, for example in the case of the product, that every tuple is in the limit as opposed to just those that satisfy the coherence condition.

[1]The category of objects and identity arrows

Powers and Copowers Assume that our diagram $D : \mathcal{I} \to C$ is constant additionally to the category $\mathcal{I}$ being discrete and assume $D(i) = A$ for some $A \in C$. Then we have another special case where the dependent product becomes a product of the same object $\Pi_{i \in \mathcal{I}} A$ which is called the *power* and denoted by $A^{\mathcal{I}}$. Note that this is in general different from the exponential because $\mathcal{I}$ is a category, not an object. However, in **Set** the power is isomorphic to the function space $\mathcal{I} \to A$ since $\mathcal{I}$ can be viewed as a set because it is discrete[1].

Similarly, the *copower* is a special case of dependent sum when D is constant and is denoted by $\mathcal{I} \bullet A$. Moreover in **Set**, this is just the pair $\mathcal{I} \times A$ with $\mathcal{I}$ again viewed as a set.

6.1 Algebraic Data Types as Limits and Colimits

Streams as Limits Assume C is a category with all limits. The streams over A from definition (2) are obtained as the limit for a diagram. We first define the functor $F : \mathbf{Set} \to \mathbf{Set}$ by $FX = A \times X$ giving the non-recursive shape of the streams. Then we define the ω^{op}-chain of approximations represented by the diagram $D : \omega^{\mathsf{op}} \to C$ defined on object by $D(1) = 1$ and $D(n+1) = F^n 1$ and on arrows by $D(1 \leq 2) =!$ (the unique map to the terminal object) and $D(n \leq n+1) = F^n!$. The limit of this ω^{op}-chain is the type of coinductive streams as depicted below:

$$\begin{tikzcd}
 & & & \underset{\longleftarrow}{\lim} F^n 1 \arrow[dlll, "\pi_1"'] \arrow[dll, "\pi_2"'] \arrow[dl, "\pi_3"'] \arrow[dr, "\pi_{n+1}"] & & \\
1 & F1 \arrow[l, "!"] & F^2 1 \arrow[l, "F!"] & \cdots \arrow[l, "F^2!"] & F^n 1 \arrow[l, "F^{n-1}!"] & \cdots \arrow[l, "F^n 1"]
\end{tikzcd}$$

Lists as Colimits Assume C is a category with all colimits. The lists over A from definition (4) are obtained as the colimit for a diagram. We first define the functor $F : \mathbf{Set} \to \mathbf{Set}$ by $FX = 1 + A \times X$ giving the non-recursive shape of the lists. Then we define the ω-chain of approximations represented by the diagram $D : \omega \to C$ defined on object by $D(1) = 0$ and $D(n+1) = F^n 0$ and on arrows by $D(1 \leq 2) =!$ (the unique map from the initial object) and $D(n \leq n+1) = F^n!$. The colimit of this ω-chain is the type of coinductive streams as depicted below:

$$\begin{tikzcd}
 & & & \underset{\longrightarrow}{\lim} F^n 0 & & \\
0 \arrow[r, "!"'] \arrow[urrr, "\mathrm{inj}_1"] & F1 \arrow[r, "F!"'] \arrow[urr, "\mathrm{inj}_2"] & F^2 1 \arrow[r, "F^2!"'] \arrow[ur, "\mathrm{inj}_3"] & \cdots \arrow[r, "F^{n-1}!"'] & F^n 1 \arrow[r, "F^n 1"'] \arrow[ul, "\mathrm{inj}_{n+1}"'] & \cdots
\end{tikzcd}$$

[1]Notice that $\mathcal{I}$ needs to be small for the collection of objects to be a set, but again, size issues are not really an issue for the focus of these notes

7 Adjunctions

Almost every universality property comes from an adjunction[1] and certainly all the constructions seen so far are in fact an instance of an adjunction.

7.0.1 Adjunctions

Given two functors $L : \mathcal{D} \to C$ and $R : C \to \mathcal{D}$ and *adjunction* is an isomorphism of homsets

$$\lfloor\cdot\rfloor : C(LA, B) \cong \mathcal{D}(A, RB) : \lceil\cdot\rceil$$

which is furthermore natural in A and B. Here $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are the maps witnessing the isomorphism. The adjunction is usually depicted as follows

$$\begin{tikzcd} C \arrow[r, bend right, "R"'] \arrow[r, phantom, "\perp"] & \mathcal{D} \arrow[l, bend right, "L"'] \end{tikzcd}$$

We say that that L is *left adjoint* to R, and viceversa, R right adjoint to L. However, you draw the adjunction the turnstile points towards the left-adjoint as indicated by $L \vdash R$.

As a consequence of the isomorphism we have that for all $f : LA \to B$ and $g : A \to RB$ there is a correspondence of arrows:

$$\lfloor f\rfloor = g \iff f = \lceil g\rceil$$

moreover, the natural isomorphism gives rise to the *fusion laws*. For $a : A' \to A$, $b : B \to B'$, $f : LA \to B$ and $g : A \to RB$

$$R(b) \cdot \lfloor f\rfloor = \lfloor b \cdot f\rfloor \tag{7}$$

$$\lfloor f\rfloor \cdot a = \lfloor f \cdot L(a)\rfloor \tag{8}$$

$$b \cdot \lceil g\rceil = \lceil R(b) \cdot g\rceil \tag{9}$$

$$\lceil g\rceil \cdot L(a) = \lceil g \cdot a\rceil \tag{10}$$

This is really all about adjunctions. All the other definitions and constructions are equivalent to this one. Furthermore, this material is very well covered elsewhere (e.g. in Awodey's book [1]) so we will not be covering it further.

7.1 Instances of Adjunctions

7.1.1 Initial and Terminal Objects

Assume that we have a category $\mathbf{1}$ with only one object $*$ and one arrow, the identity arrow id_*. Then the universality property of the initial and terminal object can be then rephrased in terms of adjunctions

$$\begin{tikzcd} C \arrow[r, bend right, "\Delta"'] \arrow[r, phantom, "\perp"] & \mathbf{1} \arrow[l, bend right, "0"'] \arrow[r, bend right, "1"'] \arrow[r, phantom, "\perp"] & C \arrow[l, bend right, "\Delta"'] \end{tikzcd}$$

[1]More in general a universal map is the initial object in the comma category $X \downarrow F$, but that is not our concern here.

where 0 is the constant functor returning the initial object and 1 is the functor returning the terminal object while Δ is the constant functor returning the element $*$.

We can prove that if the above are indeed adjunctions then the functors 0 and 1 must given the initial and terminal object respectively. The isomorphism given by the adjunction with 0 as left adjoint is as follows:

$$\lfloor\cdot\rfloor : \hom_C(0, Y) \cong \hom_{\mathbf{1}}(*, *) : \lceil\cdot\rceil$$

while for the terminal object the adjunction is given by the following natural isomorphism:

$$\lfloor\cdot\rfloor : \hom_{\mathbf{1}}(*, *) \cong \hom_C(X, 1) : \lceil\cdot\rceil$$

Exercise 7.1. *Compute the two naturality conditions and derive the fusion laws.*

We now proceed with the universal property of products and coproducts.

7.1.2 (Co)products and Products

Similary the coproduct arises as the left adjoint of the diagonal functor $\Delta : C \to C \times C$ which is defined as $\Delta X = (X, X)$ while the product arises as the right adjoint of the functor Δ:

$$\begin{tikzcd} C \arrow[r, shift right=1.5, "\Delta"'] \arrow[r, phantom, "\perp"] & C \times C \arrow[l, shift right=1.5, "+"'] \arrow[r, shift right=1.5, "\times"'] \arrow[r, phantom, "\perp"] & C \arrow[l, shift right=1.5, "\Delta"'] \end{tikzcd}$$

Exercise 7.2. *Derive the fusion laws and conclude these are exactly those of the product and coproduct respectively.*

7.1.3 Exponentials

The exponential is given by the right adjoint of the functor $(- \times A)$. That is the functor $(-)^A$:

$$\begin{tikzcd} C \arrow[r, shift right=1.5, "(-)^A"'] \arrow[r, phantom, "\perp"] & C \arrow[l, shift right=1.5, "(-)\times A"'] \end{tikzcd}$$

The isomorphism induced by this adjunction is stated as follows:

$$\lfloor\cdot\rfloor : \hom_C(X \times A, Y) \cong \hom_C(X, Y^A) : \lceil\cdot\rceil$$

Notice that here $\lfloor\cdot\rfloor$ and $\lceil\cdot\rceil$ are respectively the curry and uncurry operations in functional programming.

7.1.4 (Co)Limits

Limits and colimits stem from adjunctions too.

The limit, in particular, extends to a functor from the category of diagrams C^I to the category C simply taking a diagram and returning its limit. This functor is right adjoint to the diagonal functor $\Delta : C \to C^I$ defined as $\Delta X I = X$ mapping an object to the constant diagram which returns that object. Dually, colimits are left-adjoints to the diagonal functor. The situation is depicted below:

$$\begin{tikzcd} C \arrow[r, shift right, "\Delta"'] & C^I \arrow[l, shift right, "\overset{\textstyle \lim}{\rightarrow}"'] \arrow[l, phantom, "\perp"] \arrow[r, shift right, "\underset{\textstyle \leftarrow}{\lim}"'] & C \arrow[l, shift right, "\Delta"'] \arrow[l, phantom, "\perp"] \end{tikzcd}$$

At this point the reader should notice that the similarity between initial and terminal, coproducts and products and colimits and limits. Indeed initial objects and coproducts are special cases of colimits while terminal objects and products are special cases of limits.

7.2 Semantics of Predicative Polymorphism

Polymorphism is a core feature of most programming languages allowing developers to craft generic programs which are agnostic to the specifics of the types.

In programming languages like C, we can define algorithms that work for lists that contain different types, but we have to define the algorithm for each and everyone of the specific type we want to handle. This type of polymorphism is called *ad-hoc polymorphism*. *Predicative polymorphism* allows the programmer to write one function that is *parametric* on the type or, in words, it takes a generic type with the promise that the data of that type will not be inspected. Therefore, this kind of algorithm can be said to be agnostic as to the type of data structure given. This allows for code reusability since a polymorphic program can be written only once and can work at all types.

This flexibility provides a layer of confidence regarding the properties of the code. Philip Wadler's paper [11] offers a comprehensive exploration of these properties.

As an example, in a total language there is no program of type $\forall\alpha.\alpha$ since such a program would need to yield a value of an arbitrary type α. Consequently, $\forall\alpha.\alpha$ can be viewed as the empty type 0, which, in a total language[1], acts as the initial object.

Similarly, $\forall\alpha.\alpha \to \alpha$ has only one inhabitant, namely the identity function $\Lambda\alpha.\lambda x.x$, with Λ abstracting a type variable and λ abstracting a term variable.

Now the `reverse` function, reversing a list, is indeed a polymorphic function since it needs not inspecting the content of the single element in a list:

$$\forall\alpha.\text{List } \alpha \to \text{List } \alpha \tag{11}$$

This type grants universality by reversing elements without inspecting their specifics. Conversely, any sorting algorithm on lists needs to know that the inner type inside the list as some kind of ordering associated with it, rendering (11) unsuitable for quicksort or mergesort, for instance.

Once we defined a polymorphic function, we have the liberty to instantiate it with any desired type, even itself. This style of polymorphism is called *impredicative* and was first introduced by Girard in 1972 [2] in the context of proof theory and then proposed by Reynolds in 1983 [9] as a programming language known was *System F* or *second-order λ-calculus*.

[1]There are of course issues with non-termination but we will not address them here

For instance, the `reverse` function might operate on elements of type $\mathbb{N}$ or η, but also on the type (11) itself.

While impredicative polymorphism is convenient in programming languages, it poses a significant foundational problem when seeking for a semantic model as we did in Section 4.4. As we have seen the task of seeking a denotational model is to find a universe of sets $\mathcal{U}$ and interpreting types with n free variables as maps $\mathcal{U}^n \to \mathcal{U}$. Assuming no free variables, we would like to interpret $\forall\alpha.\alpha$ as a set. Naively we could try to interpret it as the product of sets indexed over sets. Now, the denotation of $\forall\alpha.\alpha$ would be a set for the interpretation to work and, as a result, we would have that the product of all sets would be the *set containing all sets* which is a paradoxical statement known as the Russell's paradox, suggesting that such a set cannot exist. This fact was discovered by Reynolds in 1984 [10].

Models of impredicative polymorphism have eventually been found by Pitts [8] who showed this in constructive set theory.

In these notes, we are going to avoid this problem and restrict ourselves to *predicative polymorphism* which is a variant of impredicative polymorphism where polymorphic types (*polytypes* or *type schemes*) can only be instantiated with non-polymorphic types (*monotypes*).

7.2.1 Syntax

The language we are going to define is called ML_0 . We first define the syntax by defining the set of monotypes, the set of polytypes, the set of terms and the typing judgment relation. The set of types monotypes and polytypes is defined inductively as follows:

$$\begin{array}{rcll}
A, B \in \mathrm{Types}_0 & ::= & \alpha & \text{(type variables)}\\
 & \mid & \mathsf{unit} & \text{(unit type)}\\
 & \mid & \mathsf{nat} & \text{(natural numbers)}\\
 & \mid & A \times B & \text{(products)}\\
 & \mid & A \to B & \text{(functions)}\\
\tau \in \mathrm{Types}_1 & ::= & A \mid \forall\alpha.\tau & \text{(predicative polymorphism)}
\end{array}$$

Notice that polytypes are of the form $\forall\alpha.\forall\beta.A$ where A is a monotype. Thus types of the form $\forall\alpha.(\forall\beta.\beta) \to \alpha$ for example are not allowed.

The set of λ-terms Terms is inductively defined as follows:

$$\begin{array}{rcll}
t \in \mathrm{Terms} & ::= & x & \text{(terms variables)}\\
 & \mid & () & \text{(unit)}\\
 & \mid & \underline{n} & \text{(natural numbers)}\\
 & \mid & (t_1, t_2) \mid \mathrm{prj}_1(t) \mid \mathrm{prj}_2(t) & \text{(products)}\\
 & \mid & \lambda x.t \mid t_1 t_2 & \text{(functions)}\\
 & \mid & \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 \mid \Lambda\alpha.t \mid t@A & \text{(polymorphism)}
\end{array}$$

where Λ is a binder which abstracts over type variables, the `let` binds a program M of a polytype inside a program N which returns a monotype. This allows the programmer to write programs such as

$$\mathtt{let}\ x = id\ \mathtt{in}\ x@\mathsf{unit} : \mathsf{unit} \to \mathsf{unit}$$

where $id : \forall\alpha.\alpha \to \alpha$ and the constructor $M@A$ is the application for terms that have a polytype as a type.

Note that since we have defined types using to separate layers for grammars we can constrain the types that we are going to apply to polymorphic programs. In fact, in the constructor $M@A$ only monotypes are allowed.

We first define a judgment for the contexts and the types. Technically, there are two judgments for types, one for the monotypes and one for the polytypes, but we adopt the same notation for both of them as it should be clear from the context which judgment we mean. First a type context is a list of variables. Theunit type can be typed in any well-formed context and and a type variable can by typed in any well-formed context that contains it. Finally the polymorphic types $\forall\alpha.\tau$ are well-typed if the body τ is open in α and well-typed: The context for term variables is instead a list of pairs $x : \tau$ implying that variables are of poly-typed and therefore can also be mono-typed.

$$\Gamma = (x_1 : \tau_1, \dots, x_n : \tau_n)$$

We now define the typing judgment for terms. Again, there are technically two typing judgments, the first for mono-typed programs and the second for poly-typed programs. Once again, it should be clear from the context which judgment we are using:

$$\frac{(x : \tau) \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash \underline{n} : \mathsf{nat}}$$

$$\frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathrm{prj}_1(t) : A} \qquad \frac{\Gamma \vdash t : A \times B}{\Gamma \vdash \mathrm{prj}_2(t) : B} \qquad \frac{\Gamma \vdash t_1 : A \quad \Theta \mid \Gamma \vdash t_2 : B}{\Gamma \vdash \langle t_1, t_2 \rangle : A \times B}$$

$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

$$\frac{\Gamma \vdash t_1 : \tau \qquad \Gamma, x : \tau \vdash t_2 : A}{\Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : A}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \Lambda\alpha.t : \Pi\alpha.\tau}\text{if } \alpha \notin \mathrm{Ftv}(\Gamma) \qquad \frac{\Gamma \vdash t : \Pi\alpha.\tau}{\Gamma \vdash t@A : \tau[A/\alpha]}$$

The first typing rule is for variables. Notice once again that variables can be typed with a poly-type and therefore the context should accommodate that. Additionally the polytype itself has to be well-typed in the context Θ.

We now justify some choices behind the type system.

First notice the variable rule which says that variables' types can be open. The first example that justifies this is the polymorphic identity function which is typed as follows:

$$\frac{\dfrac{\dfrac{\dfrac{}{x : \alpha \vdash x : \alpha}}{\vdash \lambda x.x : \alpha \to \alpha}}{\vdash \lambda x.x : \Pi\alpha.\alpha \to \alpha} \qquad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash id : \Pi\alpha.\alpha \to \alpha}}{\Gamma \vdash id : (\alpha \to \alpha) \to \alpha \to \alpha} \qquad \dfrac{\dfrac{}{\Gamma \vdash id : \Pi\alpha.\alpha \to \alpha}}{\Gamma \vdash id : \alpha \to \alpha}}{\Gamma \vdash id\ id : \alpha \to \alpha}}{\vdash \mathtt{let}\ id = \lambda x.x\ \mathtt{in}\ id\ id : \alpha \to \alpha}$$

7.2.2 Semantics

First of all we have to define the semantics of types. Since types can be open with type variables a type needs to be a mapping from a type environment to the set of types. A type environment is itself a map from the type variables to their set. However, as there is no set of sets we have to define a set containing only those sets are *small*. We call this set the *universe* of small sets and we denoted by $\mathcal{U}$. This set contains the singleton set and the set of natural numbers and it is closed under exponentials. To do this we define $\mathcal{U}_0 = \{1, \mathbb{N}\}$ and $\mathcal{U}_{n+1} = \{X^Y \mid X, Y \in \mathcal{U}_n\} \cup \mathcal{U}_n$ then the universe of monotypes is defined as $\mathcal{U} = \bigcup_{n\in\omega} \mathcal{U}_n$. We denote by $\mathcal{U}^n$ the n-fold product

$$\mathcal{U}^n \triangleq \underbrace{\mathcal{U} \times \cdots \times \mathcal{U}}_{n \text{ times}}$$

Now the interpretation of a monotype A with free type variables $\mathrm{Ftv}(A)$ is a mapping $[\![A]\!] : \mathcal{U}^{|\mathrm{Ftv}(A)|} \to \mathcal{U}$ where $|\mathrm{Ftv}(A)|$ is the number of free type variables in the monotype A. Thus given a semantic type environment $\iota \in \mathcal{U}^{|\mathrm{Ftv}(A)|}$ taking type variables to semantic monotypes we can define the semantics of types by induction on the as follows:

$$\begin{aligned}
[\![\alpha]\!]_\iota &= \iota(\alpha) \\
[\![\mathsf{unit}]\!]_\iota &= 1 \\
[\![\mathsf{nat}]\!]_\iota &= \mathbb{N} \\
[\![A \times B]\!]_\iota &= [\![A]\!]_\iota \times [\![B]\!]_\iota \\
[\![A \to B]\!]_\iota &= [\![B]\!]_\iota{}^{[\![A]\!]_\iota}
\end{aligned}$$

The semantics of polytypes are interpreted in a bigger universe such that $\mathcal{U}$ is contained in it. The category of sets would do as opposed to the Von Neumann universe used in Gunter's book [3]. We denote the inclusion functor by $J : \mathcal{U} \hookrightarrow \mathbf{Set}$. Now for a polytype τ as a map $[\![\tau]\!] : \mathcal{U}^{|\mathrm{Ftv}(\tau)|} \to \mathbf{Set}$. Specifically, we interpret the universal quantifier as the limit over the monotypes in $\mathcal{U}$. This is a Π-type since the universe $\mathcal{U}$ is a set with no arrows and therefore can be regarded as a discrete category as we have shown in Section 6.

$$[\![\Pi\alpha.\tau]\!]_\iota = \Pi_{X\in\mathcal{U}} [\![\tau]\!]_{\iota[\alpha\mapsto X]}$$

Recall that the Π-type is right adjoint to the diagonal functor, in this instance the base category is $\mathbf{Set}^{\mathcal{U}^n}$ with $n+1$ being the free type variables in τ.

$$\begin{tikzcd}
\mathbf{Set}^{\mathcal{U}^n \mathcal{U}} \arrow[r, bend right, "\Pi"'] \arrow[r, phantom, "\perp"] & \mathbf{Set}^{\mathcal{U}^n} \arrow[l, bend right, "\Delta"']
\end{tikzcd}$$

Notice that $[\![\tau]\!]$ lives in the category $\mathbf{Set}^{\mathcal{U}^n \mathcal{U}}$ which is the same as the category $\mathbf{Set}^{\mathcal{U}^{n+1}}$ of semantic types on $n+1$ variables.

The isomorphism induced by the adjunction is therefore as follows:

$$\lfloor\cdot\rfloor : \mathbf{Set}^{\mathcal{U}^{n+1}}(\Delta\Gamma, \tau) \cong \mathbf{Set}^{\mathcal{U}^n}(\Gamma, \Pi\tau) : \lceil\cdot\rceil \tag{12}$$

Notice that the homset in the category $\mathbf{Set}^{\mathcal{U}^n}$ is the set of natural transformations between functors of type $\mathcal{U}^n \to \mathbf{Set}$.

We now have to define the semantics of a context for term variables $\Gamma \equiv x_1 : \tau_1, \ldots, x_n : \tau_n$. Assume m is the number of free type variables in Γ. Then $\llbracket \Gamma \rrbracket$ is a object in $\mathbf{Set}^{\mathcal{U}^m}$:

$$\llbracket \Gamma \rrbracket = \llbracket T_1 \rrbracket \times \cdots \times \llbracket T_n \rrbracket$$

where the product is the defined point-wise as $(X \times Y)(\iota) = X_\iota \times Y_\iota$ for a type environment $\iota \in \mathcal{U}^m$.

The interpretation of well-typed terms $\Gamma \vdash t : \tau$ is an arrow $\llbracket \Gamma \vdash t : \tau \rrbracket : \llbracket \Gamma \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket \tau \rrbracket$ in $\mathbf{Set}^{\mathcal{U}^n}$ whereas a well-typed term $\Gamma \vdash t : A$ is an arrow $\llbracket \Gamma \vdash t : A \rrbracket : \llbracket \Gamma \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket A \rrbracket$. The interpretation is then given by induction on the typing judgment:

$$\begin{aligned}
\llbracket \Gamma \vdash () : \mathsf{unit} \rrbracket &= ! \\
\llbracket \Gamma \vdash \underline{n} : \mathsf{nat} \rrbracket &= n \\
\llbracket \Gamma \vdash x : A \rrbracket &= \pi_x \\
\llbracket \Gamma \vdash \mathrm{prj}_1(t) : A \rrbracket &= \pi_1 \circ \llbracket t \rrbracket \\
\llbracket \Gamma \vdash \mathrm{prj}_2(t) : B \rrbracket &= \pi_2 \circ \llbracket t \rrbracket \\
\llbracket \Gamma \vdash (t_1, t_2) : A \times B \rrbracket &= \langle \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket \rangle \\
\llbracket \Gamma \vdash \lambda x.t : A \to B \rrbracket &= \Lambda \llbracket t \rrbracket \\
\llbracket \Gamma \vdash t_1 t_2 : B \rrbracket &= \epsilon \circ \langle \llbracket M \rrbracket, \llbracket N \rrbracket \rangle \\
\llbracket \Gamma \vdash \mathtt{let}\ x = t_1\ \mathtt{in}\ t_2 : A \rrbracket &= \llbracket t_2 \rrbracket \circ \langle id_\Gamma, \llbracket t_1 \rrbracket \rangle \\
\llbracket \Gamma \vdash \Lambda \alpha.t : \forall \alpha.\tau \rrbracket &= \lfloor \llbracket t \rrbracket \rfloor \\
\llbracket \Gamma \vdash t@A : \tau[A/\alpha] \rrbracket &= \pi_A \circ \llbracket t \rrbracket
\end{aligned}$$

Most of the terms are interpreted as in Section 4.4. Assuming a map $\llbracket t \rrbracket : \Delta \llbracket \Gamma \rrbracket \to \llbracket \tau \rrbracket$ in $\mathbf{Set}^{\mathcal{U}^{n+1}}$ $\alpha \notin \mathrm{Ftv}(\Gamma)$ the interpretation of the introduction rule of the $\forall$ quantifier is given by the adjunction (12) defined above which is an arrow of type

$$\llbracket \Gamma \rrbracket_\iota \to \Pi_{X \in \mathcal{U}} \llbracket \tau \rrbracket_{\iota[\alpha \mapsto X]}$$

for $\iota \in \mathcal{U}^n$.

8 Monads

In this section we are going to take a closer look at computational effects and how they can be interpreted into a category. To do this need the notion of monad. Here we assume the reader has at least heard of what a monad is from functional programming. Monads in category theory are the same concept, but originally they have been given a different definition.

Definition 8.1 (Monad)**.** *For a category C, a* monad *is an endofunctor $T : C \to C$ such that there exists two natural transformations $\eta : Id \stackrel{.}{\to} T$ and $\mu : TT \stackrel{.}{\to} T$ such* *that the following diagrams hold:*

$$\begin{tikzcd}
T \arrow[r, "\eta_T"] \arrow[dr, "id_T"'] & T^2 \arrow[d, "\mu_T"] & T \arrow[l, "T\eta"'] \arrow[dl, "id_T"] \\
& T &
\end{tikzcd}
\qquad
\begin{tikzcd}
T^3 \arrow[r, "\mu_T"] \arrow[d, "T\mu"'] & T^2 \arrow[d, "\mu"] \\
T^2 \arrow[r, "\mu"'] & T
\end{tikzcd}$$

Example 8.1 (List Monad)**.** *The list type $TX = 1+A\times TX$ is a monad with $\eta_X : X \to TX$ being the map constructing a singleton list and the multiplication $\mu_X : TTX \to X$ given by concatenation applied to lists of lists.*

Example 8.2 (State Monad)**.** *Assume a set of state S and let T : **Set** $\to$ **Set** be the* state monad *$TX = S \to X \times S$, that is the monad of computations that take a state $\sigma \in S$ and returns an output in A along with the modified state. The unit of the monad is given be $a \mapsto \lambda\sigma.\langle a, \sigma\rangle$ and the multiplication is given by*

$$c \mapsto \lambda\sigma.c'(\sigma') \text{ where } (c', \sigma') = c(\sigma)$$

For the reader more familiar with functional programming, the multiplication gives raise to the bind operation $¿¿=_A : TA \to (A \to TB) \to TB$ defined by $\mu_A \circ T(f)$.

8.0.1 Adjunctions determine Monads

Given a pair of adjoint functors $L \vdash R$ we can construct both a monad, given by RL and a comonad, given by LR. The unit of the monad and the counit of the comonad are defined as follows

$$\eta_A = \lfloor id_{LA} \rfloor$$
$$\epsilon_B = \lceil id_{RB} \rceil$$

The join or multiplication of the monad $\mu : RLRL \to RL$ is defined as $\mu = R\epsilon_L$ and the cojoin or comultiplication $\delta : LR \to LRLR$ is defined as $\delta = L\eta_R$. The operations of the comonad are dually defined.

8.1 The Kleisli Category

The Kleisli category is the category of arrows that produce an effect T.

Definition 8.2 (Klesli Category)**.** *For a category C and a monad (T, η, μ) the Kleisli category, denoted by C_T, is the category where objects are the objects in C and arrows $f : A \to B$ are arrows $f_T : A \to TB$ in C.*

We would like to stress that when we speak about arrows $f : A \to B$ in the Kleisli category C_T we mean arrows $f_T : A \to TB$ in C. We have to prove that C_T is a category. This is easy to check as for every object A we have and identity arrow $id_A : A \to A$ given by the unit of the monad $\eta_A : A \to TA$ and for arrows $f : A \to B$ and $g : B \to C$ in C_T we can construct the composite $g \circ f : A \to C$ using the functorial action of the monad $T(g)$ and the multiplication of the monad μ_C:

$$\begin{tikzcd}
A \arrow[r, "f"] \arrow[rrr, bend right, "(g\circ f)_T"'] & TB \arrow[r, "T(g)"] & T^2C \arrow[r, "\mu_C"] & TC
\end{tikzcd}$$

8.2 The Computational λ-calculus

The computational λ-calculus which we denote by λ_C is a calculus with effects first devised by Eugenio Moggi [7] who was the first to discover the connection between computational effects and monads.

In this section we define the computational λ-calculus and we give it an interpretation in the Kleisli category $\mathcal{C}_T$ for a (strong) monad T.

8.2.1 Syntax

We first define the syntax of λ_C by defining the set of types, the terms and the typing system as usual. To demonstrate the utility of monads we are only going to need basic types and function spaces:

$$\begin{array}{lcll} A, B \in \text{Types} & ::= & \mathsf{unit} & \text{(unit type)} \\ & \mid & A \to B & \text{(functions)} \end{array}$$

while the set of λ-terms Terms is inductively defined as follows

$$\begin{array}{lcll} t \in \text{Terms} & ::= & x & \text{(terms variables)} \\ & \mid & \text{bang} & \text{(effects)} \\ & \mid & \lambda x.t \mid t_1 t_2 & \text{(functions)} \end{array}$$

where bang is a program producing an effect. The typing judgement relation $\vdash$ inductively as follows

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash \text{bang} : \mathsf{unit}}$$
$$\frac{\Gamma \vdash t_1 : A \to B \qquad \Gamma \vdash t_2 : A}{\Gamma \vdash t_1 t_2 : B} \qquad \frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \lambda x.t : A \to B}$$

Note that the program bang returns nothing so we type it with the type unit.

8.2.2 Semantics

We interpret the context Γ as usual:

$$[\![x_1 : A_1, \cdots, x_n : A_n]\!] = [\![A_1]\!] \times \cdots \times [\![A_n]\!]$$

The interpretation of types is then a function $[\![\,]\!]$: Types $\to$ Obj$(\mathcal{C}_T)$. Notice that Obj$(\mathcal{C}_T)$ is exactly Obj$(\mathcal{C})$.

$$[\![\mathsf{unit}]\!] = 1$$
$$[\![A \to B]\!] = T[\![B]\!]^{[\![A]\!]}$$

The placement of which types can produce an effect is crucial. When dealing with effects it is customary to prefer a call-by-value semantics to avoid that effectul computation passed onto functions as inputs could be executed more than once to the nature of call-by-name.

Because in call-by-value effects are computed before the β-reduction rule applies there is no need for input values to be computations, they are actually normalised values. However, once an input value is applied to a function, this can produce an effect.

We now interpret the terms of the language by induction on the typing judgment. For a well-typed term $\Gamma \vdash t : A$ we give an arrow $[\![t]\!]$ of type $[\![\Gamma]\!] \to [\![A]\!]$ in $\mathcal{C}_T$ as usual, but here the reader should keep in mind that this is really a map of type $[\![\Gamma]\!] \to T[\![A]\!]$ in $\mathcal{C}$

$$\begin{aligned}
[\![\Gamma \vdash () : \mathsf{unit}]\!] &= ! \\
[\![\Gamma \vdash \text{bang} : \mathsf{unit}]\!] &= b \\
[\![\Gamma \vdash x : A]\!] &= \eta_{[\![A]\!]} \circ \pi_x \\
[\![\Gamma \vdash \lambda x.t : A \to B]\!] &= \eta_{T[\![A]\!]^{[\![B]\!]}} \circ \Lambda[\![t]\!] \\
[\![\Gamma \vdash t_1 t_2 : B]\!] &= \text{app}([\![t_1]\!], [\![t_2]\!])
\end{aligned}$$

In order to explain the interpretation we work in the category $\mathcal{C}$ so the application of the monad T is explicit. We also remove the semantic brackets for the sake of removing clutter. Now, for λ-abstraction we work as follows. Assume a map $t : \Gamma \times A \to TB$. We have to define a map $\Gamma \to T(TB^A)$. By currying t we obtain $\Lambda t : \Gamma TB^A$ and by post-composing this map with η_{TB^A} we obtain the type $T(TB^A)$.

Function application is more tricky in that this is the point where we need the monad to be strong. For two maps $t_1 : \Gamma \to T(TB^A)$ and $t_2 : \Gamma \to TA$ We define the map $app(t_1, t_2) : \Gamma \to TB$ as follows:

$$\begin{tikzcd}
\Gamma \arrow[rr, "\text{app}(t_1,t_2)"'] \arrow[d, "\langle t_1,t_2\rangle"] & & TB \\
T(TB^A) \times TA \arrow[d, "st_{TB^A,TA}"] & & T^2B \arrow[u, "\mu_B"] \\
T(TB^A \times TA) \arrow[r, "T(st_{TB^A,A})"] & T(T(TB^A \times A)) \arrow[r, "T^2(\epsilon)"] & T^3B \arrow[u, "\mu_{TB}"]
\end{tikzcd}$$

References

[1] Steve Awodey. *Category Theory*. Oxford University Press, Inc., USA, 2nd edition, 2010.

[2] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse d'État, Université Paris VII, 1972.

[3] C.A. Gunter. *Semantics of Programming Languages: Structures and Techniques*. Foundations of computing. MIT Press, 1992.

[4] Saunders MacLane. *Categories for the Working Mathematician*. Springer-Verlag, New York, 1971. Graduate Texts in Mathematics, Vol. 5.

[5] Alfio Martini. Category theory and the simply-typed lambda-calculus. 1996.

[6] B. Milewski. *Category Theory for Programmers*. Blurb, Incorporated, 2018.

[7] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991.

[8] Andrew M. Pitts. Polymorphism is set theoretic, constructively. In David H. Pitt, Axel Poigné, and David E. Rydeheard, editors, *Category Theory and Computer Science, Edinburgh, UK, September 7-9, 1987, Proceedings*, volume 283 of *Lecture Notes in Computer Science*, pages 12–39. Springer, 1987.

[9] John C. Reynolds. Types, abstraction and parametric polymorphism. In R. E. A. Mason, editor, *Information Processing 83, Proceedings of the IFIP 9th World Computer Congress, Paris, France, September 19-23, 1983*, pages 513–523. North-Holland/IFIP, 1983.

[10] John C. Reynolds. Polymorphism is not set-theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984.

[11] Philip Wadler. Theorems for free! In Joseph E. Stoy, editor, *Proceedings of the fourth international conference on Functional programming languages and computer architecture, FPCA 1989, London, UK, September 11-13, 1989*, pages 347–359. ACM, 1989.