Category Theory Course Notes

Marco Paviotti

December 19, 2025

Contents

1 Introduction **2**

2 Elements of Set Theory **4**
2.1 Relations and Functions 4
2.2 Cardinality and Isomorphism 5
2.3 Indexed Families of Sets 5
2.4 The Russel's Paradox 5
2.4.1 The Axiom of Regularity (Foundation) 6
2.5 Exercises 7

3 Categories, Functors and Natural Transformations **7**
3.1 Categories 8
3.2 Isomorphisms 8
3.3 Functors 9
3.4 Natural Transformations 11
3.5 Constructions on Categories 11
3.6 The Homset Functor 12
3.7 Equivalence of Categories 13

4 Universal Objects **14**
4.1 Initial and terminal objects 14
4.2 The Natural Numbers object 15
4.3 Products, CoProducts and Exponentials 17
4.3.1 Products 17
4.3.2 Coproducts 19
4.3.3 Exponentials 20
4.4 Exercises 22

5 Semantics of the Simply Typed λ-Calculus **24**
5.1 Syntax 25
5.2 Semantics 25
5.3 Exercises 27

6 Limits and Colimits **27**
6.1 Limits . . . 27
6.2 Colimits . . . 29
6.3 The Yoneda Lemma . . . 31
6.3.1 The Yoneda Embedding . . . 32
6.4 Exercises . . . 33

7 Data Types as Fixed-Points **34**
7.1 Least Fixed-Points as Colimits . . . 34
7.2 Greatest Fixed-Points as Limits . . . 35

8 Adjunctions **36**
8.1 Adjunctions . . . 36
8.2 Initial and Terminal Objects . . . 37
8.3 (Co)products and Products . . . 38
8.4 Exponentials . . . 38
8.5 (Co)Limits . . . 38

9 Semantics of Predicative Polymorphism **39**
9.1 Syntax . . . 40
9.2 Semantics . . . 41

10 Monads **43**
10.1 Adjunctions determine Monads . . . 44
10.2 The Kleisli Category . . . 44

11 The Computational λ-calculus **44**
11.1 Syntax . . . 45
11.2 Semantics . . . 45

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 [5].

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].

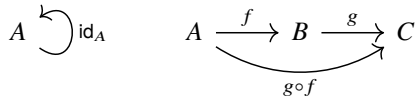
In these notes, we will try to convince the reader that category theory covers a wide variety of subjects; from logic to type theory, hence programming, algebra and topology and it is therefore a subject worth studying for the reader's own good.

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 $\mathsf{id}_A : A \to A$ for every object. The second, *composition*, embodies transitivity by allowing morphisms $f : A \to B$ and $g : B \to C$ to compose into $g \circ f : A \to C$.



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 summarise 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 $b \in B$. Given an equivalence relation $\sim$ on X, the *quotient* of a set X by an equivalence relation $\sim$ is the set of equivalence classes:

$$X/{\sim} = \{[x] \mid x \in X\},$$

where $[x] = \{y \in X \mid x \sim y\}$. The quotient map $q : X \to X/{\sim}$ sends each element $x \in X$ to its equivalence class $[x]$. The quotient set partitions X into disjoint equivalence classes. The union of two sets A and B can be expressed as a quotiented disjoint union:

$$A \cup B = (A \uplus B)/{\sim},$$

where $(1, a) \sim (2, b)$ if and only if $a = b$.

2.1 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 $A \to B$.

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.

An *equivalence relation* on a set X is a relation R that satisfies three properties: *reflexivity*, meaning that every element is related to itself ($\forall x \in X, xRx$); *symmetry*, meaning that if one element is related to another, then the second is related to the first

$(\forall x, y \in X, xRy \Rightarrow yRx)$; and *transitivity*, meaning that if one element is related to a second and the second to a third, then the first must be related to the third $(\forall x, y, z \in X, xRy$ and $yRz \Rightarrow xRz)$.

A *preorder* is a relation that satisfies only reflexivity and transitivity, making it a weaker form of ordering that allows for indistinguishable elements. A *partial order* strengthens this by also requiring *antisymmetry*, which states that if two elements are mutually related, they must be equal $(\forall x, y \in X, xRy$ and $yRx \Rightarrow x = y)$.

2.2 Cardinality and Isomorphism

For two sets X, Y, a *function* $f : X \to Y$ is called *injective* (or one-to-one) if different inputs always produce different outputs, formally expressed as $f(x_1) = f(x_2) \Rightarrow x_1 = x_2$. It is *surjective* (or onto) if every element of Y is mapped to by at least one element of X, meaning $\forall y \in Y, \exists x \in X$ such that $f(x) = y$. When a function is both injective and surjective, it is called a *bijection* (or a one-to-one correspondence), ensuring that every element of X is uniquely paired with an element of Y and vice versa, allowing for the existence of an inverse function $f^{-1} : Y \to X$.

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 $f : A \to B$ with an inverse $f^{-1} : B \to A$ 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 $A = \{a_1, a_2, \ldots, a_n\}$. A set is *infinite* if it is not finite. A set is *countable* if it is isomorphic to the natural numbers $\mathbb{N}$.

2.3 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 $i \in I$:

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

2.4 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.

2.4.1 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.

2.5 Exercises

1. Define the set $A = \{x \in \mathbb{Z} \mid x^2 < 25\}$ explicitly by listing all its elements.
2. Consider the set $B = \{(x, y) \in \mathbb{Z}^2 \mid x + y = 5\}$. List five elements of B.
3. Let R be a relation on $\mathbb{Z}$ defined by xRy if and only if $x - y$ is even. Prove that R is an equivalence relation.
4. Define a relation S on $\mathbb{R}$ by xSy if and only if $xy \geq 0$. Determine whether S is reflexive, symmetric, and/or transitive.
5. Consider the relation T on the power set $\mathcal{P}(\mathbb{N})$ defined by ATB if and only if $A \subseteq B$. Is T a partial order?
6. Define a function $f : \mathbb{Z} \to \mathbb{Z}$ by $f(x) = x^2 - 3x + 2$. Determine whether f is injective and/or surjective.
7. Consider the function $g : \mathbb{Q} \to \mathbb{Q}$ defined by $g(x) = 2x + 1$. Show that g is a bijection and find its inverse.
8. Let $h : \mathbb{N} \to \mathbb{N}$ be defined by

$$h(n) = \begin{cases} n/2, & \text{if } n \text{ is even} \\ 3n + 1, & \text{if } n \text{ is odd} \end{cases}$$

   Compute $h(1), h(2), h(3), h(4), h(5)$.
9. Prove that the interval $(0, 1)$ and $(0, 2)$ in $\mathbb{R}$ have the same cardinality.
10. Show that $\mathbb{N}$ and $\mathbb{Z}$ have the same cardinality by constructing an explicit bijection.
11. Prove that the set of all finite binary strings is countable, while the set of all infinite binary sequences is uncountable.
12. Let $A_n = \{x \in \mathbb{R} \mid x > \frac{1}{n}\}$ for $n \in \mathbb{N}$. Find $\bigcap_{n=1}^{\infty} A_n$.
13. Consider the sequence of sets $B_n = \{0, 1, 2, \ldots, n\}$. Determine $\bigcup_{n=1}^{\infty} B_n$ and $\bigcap_{n=1}^{\infty} B_n$.
14. Compute the Cartesian product

$$\prod_{i=1}^{\infty} \{0, 1\}$$

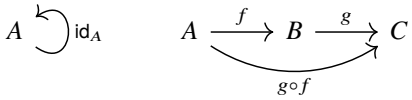
    and determine whether it is countable or uncountable.

3 Categories, Functors and Natural Transformations

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.

3.1 Categories

Definition 3.1 (Category)**.** *A* category C *is a collection of objects* $A, B, C \ldots$ *denoted by Obj(C) and a set of arrows* $f, g, h \ldots$ *denoted by Arr(C). Additionally, for each object A there exists an identity arrow* $\mathsf{id}_A : A \to A$ *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:*



Additionally, these arrows obey the identity and associativity laws:

$$\mathsf{id}_A \circ f = f \circ \mathsf{id}_A = f$$
$$f \circ (g \circ h) = (f \circ g) \circ h$$

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** and monotone functions, 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 2.4. 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*.

3.2 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 $f : A \to B$ 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:

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 $f : A \to B$ that has an inverse $g : B \to A$ 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.

3.3 Functors

Imagine the you have some object A representing the type for some data and a data type which wraps the elements of A around some structure. For example, this structure may be the lists over A, written $\mathsf{List}\ A$. Graphically, we may draw a list of As as a box $\boxed{\text{A}}$ with inside some piece of data of type A. Now given a map $f : A \to B$ we can recursively go through the list an change the data inside using f obtaining a map, which we call $\mathsf{List}\ f$, of type

$$\boxed{\text{A}} \to \boxed{\text{B}}$$

transforming the data inside a list while preserving the structure of the list. Generalising, we can say the box is a *functor* F and the map $F(f)$ is the *functorial action* of F.

Another way of looking at functors is as order-preserving maps between categories. We covered briefly the concept of preorders as particular sets with an additional structure. In a preorder $(X, \leq)$, elements have a sense of order: if $a \leq b$, then a comes before b. A *structure-preserving* map between preorders is called a *monotone function*. This map respects the order of the set X in the following way: if $a \leq b$ in one preorder, then their images still satisfy $f(a) \leq f(b)$ in the other.

Categories are a richer version of a preorder, where instead of just "$\leq$", we have morphisms (arrows) between objects.

A *functor* is a map between categories that, just like a monotone function, preserves structure. It has two components:

- It takes each object X in one category and assigns it to an object $F(X)$ in another.
- It takes each morphism $f : X \to Y$ and assigns it to a morphism $F(f) : F(X) \to F(Y)$, keeping the relationships intact.

with the additional conditions that that it also also respects how morphisms behave, that is it respects identity morphisms and it respects composition. The formal definition follows:

Definition 3.3 (Functor)**.** *Let $\mathcal{C}$ and $\mathcal{D}$ be two categories. A* functor $F : \mathcal{C} \to \mathcal{D}$ *is a mapping between categories that associates objects in $\mathcal{C}$ to an object $F\mathcal{C} \in Obj(\mathcal{C})$ 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:*

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

We are going to look at some examples in great depth whilst we are going to leave others to the reader to look at.

A very popular example of a functor is the "maybe" functor

$$FX = X + 1$$

This can be viewed as the type of computations that either return an element of X, written $(\mathsf{Just}\ x)$ or fail, written $\mathsf{Nothing}$. In the former case is equivalent to write $\mathtt{inl}(x)$, in the latter case this is equivalent to writing $\mathtt{inr}(*)$. The functorial action is defined as $F(f) = f + 1$ which is a map of type $X + 1 \to Y + 1$ by applying f to the left part of the coproduct. This is formally defined using the unique mapping property of the coproduct (left as exercise).

Another very popular functor is list functor defined as the least solution to the following domain equation

$$\mathsf{List}\ A \cong 1 + A \times \mathsf{List}\ A$$

where $\phi : 1 + A \times \mathsf{List}\ A \to \mathsf{List}\ A$ is an isomorphism. This is recursively defined using a technique we are going to see later in this notes, for now we are just going to assume this is a well-defined inductive definition, i.e. that this isomorphism exists. The empty list `[]` is represented by the element $\phi \circ \iota_1$, the cons operation `(::)` is defined by $\phi \circ \iota_2$.

The functorial action is defined by $\mathsf{List}\ f = 1 + f \times \mathsf{List}\ A$, using the functorial action of the coproduct and the product.

Finally, the stream functor is defined as the greatest solution to the following equation

$$\mathsf{Str}\ A \cong A \times \mathsf{Str}\ A \tag{1}$$

where ϕ is the operation $\mathsf{cons} : A \times \mathsf{Str}\ A \to \mathsf{Str}\ A$, the head $\mathsf{head} : \mathsf{Str}\ A \to A$ is defined as $\pi_1 \circ \phi^{-1}$ and the tail $\mathsf{tail} : \mathsf{Str}\ A \to \mathsf{Str}\ A$ is defined as $\pi_2 \circ \phi^{-1}$.

The functorial action is defined by $\mathsf{Str}\ (f) = f \times \mathsf{Str}\ A$ using the functorial action of the coproduct and the product.

We list some more examples of functors, the interested reader is invited to work out the details of these definitions.

Example 3.1. *Some popular functors include:*

- *the identity functor $FX = X$ with the obvious functional action*
- *the product functor (on one variable) $FX = A \times X$ with functorial action $F(f) = A \times f$*
- *the exponential $FX = X^A$ with the functorial action given by post-composition, that is $F(f) = \Lambda(f \circ \epsilon)$ where $\epsilon : X^A \times A \to Y$ is the evaluation map*

- *the* homset functor $FX = C(A, X)$. *This will be discussed more in depth later*

From the definition of functor it follows almost directly that every functor preserves isomorphisms:

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

the converse is not true unless the functor is surjective on morphisms, that is the functor is *full*. When the functorial action is injective the functor is called *faithful*.

Proposition 3.1. *A fully faithful functor F preserves and reflects isomorphisms:*

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

Exercise 3.1. *Prove Proposition 3.1*

3.4 Natural Transformations

Suppose that you want to write an algorithm which converts a computation on X which may fail into the list of values X this computation can produce. This means we have to write a function from Maybe X to List X regardless of what X. The most natural way to do this is by mapping `Nothing` to [], `Just` x to a finite list of xs, thus one possible choice is $[x]$.

This transformation is agnostic to the type of values. Since it holds for all type of values the conversion is not arbitrary but follows a deep categorical pattern. Thus, turning Maybe into List this way is a natural transformation.

Definition 3.4 (Natural Transformation)**.** *For two functors $F, G : C \to \mathcal{D}$, 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:*

$$\begin{array}{ccc} FA & \xrightarrow{\phi_A} & GA \\ {\scriptstyle F(f)}\downarrow & & \downarrow{\scriptstyle G(f)} \\ FB & \xrightarrow{\phi_B} & GB \end{array}$$

3.5 Constructions on Categories

The *category of functors*, denoted as $[C, \mathcal{D}]$, is a category where the objects are functors $F : C \to \mathcal{D}$, where C and $\mathcal{D}$ are two categories. The morphisms in $[C, \mathcal{D}]$ are natural transformations between functors.

An isomorphism in $[C, \mathcal{D}]$ is a *natural isomorphism*, which means that for a natural transformation $\phi : F \to G$, each $\phi_X : F(X) \to G(X)$ is an isomorphism in $\mathcal{D}$. This implies that F and G are isomorphic functors in $[C, \mathcal{D}]$.

The *category of categories*, denoted **Cat**, is the category whose objects are *small categories* and whose morphisms are functors between them. A category consists of objects, morphisms (arrows), composition of morphisms that is associative, and identity morphisms. Since the collection of *all* categories is too large to form a category, **Cat** typically consists of *small categories*, meaning those whose collections of objects and

morphisms form sets rather than proper classes. The morphisms in **Cat** are *functors*, which map objects and morphisms between categories while preserving composition and identity morphisms. Functors compose naturally: if $F : \mathcal{C} \to \mathcal{D}$ and $G : \mathcal{D} \to \mathcal{E}$ are functors, their composition $G \circ F : \mathcal{C} \to \mathcal{E}$ is also a functor. The identity morphisms in **Cat** are identity functors, which map each object and morphism to itself. In addition to functors, there exist *natural transformations*, which provide structure-preserving ways to transition between functors. This makes **Cat** more than just a category – it is actually a *2-category*, where objects are categories, 1-morphisms are functors, and 2-morphisms are natural transformations between functors. This additional structure allows for deeper relationships between categories, making **Cat** fundamental in higher category theory.

Having the categories of categories allows us to talk about *isomorphisms of categories*. Following the definition of isomorphism these are the categories such that there exists a functor $F : \mathcal{C} \to \mathcal{D}$ along with an inverse functor F^{-1}.

A *product category* refers to the *cartesian product* of two categories. Given two categories $\mathcal{C}$ and $\mathcal{D}$, their product category, denoted as $\mathcal{C} \times \mathcal{D}$, is a category where objects are pairs (C, D), where C is an object from $\mathcal{C}$ and D is an object from $\mathcal{D}$. Morphisms are pairs (f, g), where $f : C \to C'$ is a morphism in $\mathcal{C}$, and $g : D \to D'$ is a morphism in $\mathcal{D}$. Composition of morphisms is defined component-wise as

$$(f', g') \circ (f, g) = (f' \circ f, g' \circ g).$$

Identity morphisms are given by $(\mathrm{id}_C, \mathrm{id}_D)$, where id_C and id_D are identity morphisms in $\mathcal{C}$ and $\mathcal{D}$, respectively.

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

For a category $\mathcal{C}$, the category $\mathcal{C}^{\mathsf{op}}$ is defined such that it has as objects the same objects as $\mathcal{C}$ and for every morphism $f : A \to B$ in $\mathcal{C}$ a morphism $f^{\mathsf{op}} : B \to A$. It follows straightforwardly that

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

The opposite category is a useful tool for exploring dualities, where a statement about $\mathcal{C}$ has a corresponding dual statement about $\mathcal{C}^{\mathrm{op}}$. We will go back to this topic when we introduce the concept of a functor in Section 3.3.

3.6 The Homset Functor

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

The first is the *covariant hom-functor* $\mathcal{C}(A, -) : \mathcal{C} \to \mathbf{Set}$, which assigns to each object B the set of morphisms $\mathcal{C}(A, B)$. Given an arrow $f : B \to B'$ in $\mathcal{C}$, the functor

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

maps it to the function $C(A, f) : C(A, B) \to C(A, B')$, which acts by *post-composition*: each arrow $g : A \to B$ is sent to $f \circ g : A \to B'$.

The second is the *contravariant hom-functor* $C(-, B) : C^{\mathsf{op}} \to \mathbf{Set}$, which assigns to each object A the set $C(A, B)$. This functor is *contravariant* because it reverses arrows: a morphism $f : A \to A'$ in C is mapped to the function $C(f, B) : C(A', B) \to C(A, B)$, which acts by *pre-composition*: each arrow $g : A' \to B$ is sent to $g \circ f : A \to B$.

Functors which are *naturally isomorphic* to the functor $C(A, -)$ for some $A \in \mathrm{Obj}(()C)$ are called *representable functors*.

For example, the stream functor $\mathtt{Str} : \mathbf{Set} \to \mathbf{Set}$ is defined as the greatest solution to the domain equation

$$\mathtt{Str}A \cong A \times \mathtt{Str}A$$

This can be seen as the type of infinite lists over A. Let ι be the isomorphism, then the head of the stream is defined by post-composition of the first projection with ι and the tail by the second projection precomposed with ι.

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

$$\mathsf{Str}\ A \cong \mathbf{Set}(\mathbb{N}, A) \tag{2}$$

This is easy to see since we can take an infinite stream s and rename its elements such that each element has the natural number attached to it representing its position within the stream

$$a_1 a_2 \ldots a_n \ldots$$

This forms obviously a map $\mathbb{N} \to A$.

3.7 Equivalence of Categories

In category theory, an *equivalence of categories* formalizes the idea that two categories, while not necessarily identical, have the same essential structure. This concept is weaker than an isomorphism of categories, which requires a strict one-to-one correspondence between objects and morphisms. Instead, an equivalence allows for structure-preserving correspondences that hold only up to isomorphism.

Definition 3.5 (Equivalence of Categories)**.** *Two categories C and $\mathcal{D}$ are said to be* equivalent *if there exist functors*

$$F : C \to \mathcal{D} \quad \textit{and} \quad G : \mathcal{D} \to C$$

together with natural isomorphisms

$$\eta : id_C \xrightarrow{\cdot} G \circ F \quad \textit{and} \quad \varepsilon : id_{\mathcal{D}} \xrightarrow{\cdot} F \circ G.$$

These isomorphisms ensure that composing the functors in either direction gives something naturally isomorphic to the identity functor on each category. In this sense, the categories C and $\mathcal{D}$ are structurally the same from the perspective of category theory.

This idea allows us to treat equivalent categories as interchangeable for most practical purposes. Although the objects and morphisms may differ formally, their categorical behavior—how morphisms compose, how limits and colimits behave, how functors interact – is preserved up to isomorphism. A classic example is the equivalence between the category of finite-dimensional vector spaces over a field and the category of finite-dimensional representations of a certain algebra. While these categories are not isomorphic, they model the same underlying mathematical structure and thus are considered categorically equivalent.

4 Universal Objects

4.1 Initial and terminal objects

Consider the false proposition in logic. When we assume something absurd we can derive anything, hence a false proposition entails any formula A. In programming this is expressed in terms of empty data types which are denoted sometimes with 0. Having a piece of data of type 0, say $x : 0$, is absurd since the empty type contains nothing, thus performing case analysis on this data will be vacuous as state in the axiom below

$$\frac{}{x : 0 \vdash \text{case } x \text{ of } \{\} : A}$$

The axiom above may look strange at first. The variable x is case analysed on but there is no term inside the case operator. But this makes sense because case analysis on a data structure means providing a program to continue with for each constructor in the data structure, however there is no constructor in the empty type so there is nothing further we need to provide.

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:

$$0 \overset{!_A}{\dashrightarrow} A$$

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.

An equivalent formulation is by viewing what happens in the homsets. In particular,

$$C(0, A) \cong \mathbf{1}(*, *)$$

natural in A. This can be proven using the uniqueness property which is more formally stated as follows:

$$f = !_A \qquad \text{(Uniqueness Property)} \tag{3}$$

for all $f : 0 \to A$. This implies a reflection and fusion law:

$$id_0 = !_0 \qquad \text{(Reflection Law)} \tag{4}$$
$$k \circ !_A = !_B \qquad \text{(Fusion Law)} \tag{5}$$

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:

$$A \overset{!_A}{\dashrightarrow} 1$$

It is important to know that such objects in category theory are unique only up-to isomorphism. Similarly to the initial object we can view this property also from the point of view of representable homsets:

$$\mathbf{1}(*, *) \cong C(A, 1)$$

natural in A. Formally, this can be derived by the uniqueness property of the terminal object can be stated as follows::

$$f = !_A \qquad \text{(Uniqueness Property)} \tag{6}$$

for all $f : A \to 1$. This implies a reflection and fusion law:

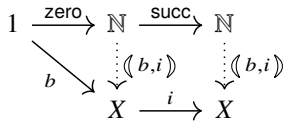
$$id_1 = !_1 \qquad \text{(Reflection Law)} \tag{7}$$
$$!_B \circ k = !_A \qquad \text{(Fusion Law)} \tag{8}$$

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.

4.2 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 $1 \xrightarrow{\text{zero}} \mathbb{N}$ and the successor morphism $\mathbb{N} \xrightarrow{\text{succ}} \mathbb{N}$ 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 $1 \xrightarrow{b} X$ (representing a base case) and $X \xrightarrow{g} X$ (representing a recursive step), there exists a unique morphism $\mathbb{N} \xrightarrow{(\!| b,i |\!)} X$ such that $(\!| b, i |\!) \circ \mathsf{zero} = b$ and $(\!| b, i |\!) \circ \mathsf{succ} = i \circ h \circ (\!| b, i |\!)$



What is important to notice is that the existence of this map provides a way of defining functions by recursion over the natural numbers, while uniqueness gives an *inductive proof principle*. To see this object from the point of view of arrows we need to first define a category of natural numbers objects over a base category C, denoted $\mathcal{N}$. That is the category there objects are objects of C with arrows $1 \xrightarrow{x_1} X \xrightarrow{x_2} X$ and arrows are arrows $f : X \to Y$ in C such that for two natural numbers objects $1 \xrightarrow{x_1} X \xrightarrow{x_2} X$ and $1 \xrightarrow{y_1} Y \xrightarrow{y_2} Y$ we have $f \circ x_1 = y_1$ and $f \circ x_2 = y_2 \circ f$. Now we can state the isomorphism of homsets

$$\mathcal{N}((\mathbb{N}, \mathsf{zero}, \mathsf{succ}), (X, x_1, x_2)) \cong C(0, X)$$

natural in X. This isomorphism is derived from the uniqueness property which is formally written as follows:

$$f = (\!| b, i |\!) \iff i = f \circ \mathsf{zero} \text{ and } f \circ \mathsf{succ} = i \circ f \tag{9}$$

for all $f : \mathbb{N} \to X$. From the uniqueness principle we can derive the following properties:

$$(\!| b, i |\!) \circ \mathsf{zero} = i \text{ and } (\!| b, i |\!) \circ \mathsf{succ} = i \circ (\!| b, i |\!) \qquad \beta\text{-law} \tag{10}$$

$$h = (\!| h \circ \mathsf{zero}, h \circ \mathsf{succ} |\!) \qquad \eta\text{-law} \tag{11}$$

$$(\!| \mathsf{zero}, \mathsf{succ} |\!) = id_{\mathbb{N}} \qquad \text{Reflection Law} \tag{12}$$

$$\text{for all } f : A \to B, b' \circ f = b \text{ and } f \circ i = i' \circ f \tag{13}$$

$$f \circ (\!| b, i |\!) = (\!| b', i' |\!) \qquad \text{Fusion Law} \tag{14}$$

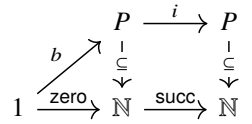
As an example, in **Set**, we can prove that all natural numbers are either even or odd by defining a subset of the natural numbers

$$P = \{n \in \mathbb{N} \mid \mathsf{even}(n) \vee \mathsf{odd}(n)\}$$

We aim to show that $P \cong \mathbb{N}$, i.e., every natural number is either even or odd. To achieve this, we construct the necessary maps that induce a unique morphism from $\mathbb{N}$ to P, which establishes the required property.

First, we define a base case function $b : 1 \to P$ by setting $b(0) = 0$, which is even and thus belongs to P. Next, we define an induction step function $i : P \to P$ by specifying that if $n \in P$, then $i(n) = n + 1$ must also be in P. We must prove that this

mapping is well-defined, that is that $n + 1$ is either even or odd. This is the crux of the proof since it corresponds to our inductive step. Obviously, if n is even then $n + 1$ is odd thus it belongs to P and similarly for when n is odd. Notice we have defined an object P along with two mappings i and b such that the following diagrams commute:



By the universal property of the natural numbers, there exists a unique map $(\!|\, [b, i] \,|\!) : \mathbb{N} \to P$ satisfying the recursive definition given by b and i. It is easy now to prove that $P \cong \mathbb{N}$ using set inclusion by showing the maps $P \subseteq \mathbb{N}$ and $(\!|\, [b, i] \,|\!) : \mathbb{N} \to P$ are mutually inverses to each other, meaning that

$$(\!|\, [b, i] \,|\!) \circ \subseteq = \mathsf{id}_P \qquad \subseteq \circ (\!|\, [b, i] \,|\!) = \mathsf{id}_{\mathbb{N}}$$

We leave this last step as an exercise.

4.3 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.

4.3.1 Products

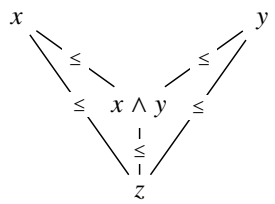
In logic, the *and* operator is introduced by stating that if Γ is a set of true propositions which entails A and this context Γ entails also B then of course $\Gamma \vdash B$. This is called the introduction rule of the product:

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

The product has also two elimination rules given by:

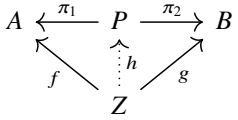
$$\frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}$$

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 $z \leq a \wedge b$.



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 $\pi_1 : P \to A$ and $\pi_2 : P \to B$ such that for any object Z with arrows $f : Z \to A$ and $g : Z \to B$ there exists a unique morphism $h : Z \to P$ making the following diagram commute:*



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

From the definition of product we can derive the following isomorphism of arrows

$$C \times C((Z, Z), (A, B)) \cong C(Z, A \times B)$$

natural in $Z, A, B : C$. Intuitively, if we have two arrows $f : Z \to A$ and $g : Z \to B$ we can form the arrow $Z \to A \times B$ and viceversa from an arrow into the product we can obtain two arrows into A and B by post-composing with the respective projections.

The isomorphism is proven by using the uniqueness property which can be more formally written as follows:

$$\pi_1 \circ g = x \text{ and } \pi_2 \circ g = y \iff \langle x, y \rangle = g \tag{15}$$

for all $g : Z \to A \times B$. This implies the following properties:

$$\pi_1 \circ \langle x, y \rangle = x \text{ and } \pi_2 \circ \langle x, y \rangle = y \qquad \beta\text{-laws} \tag{16}$$

$$\langle \pi_1 \circ h, \pi_2 \circ h \rangle = h \qquad \eta\text{-law} \tag{17}$$

$$\langle \pi_1, \pi_2 \rangle = id_{A \times B} \qquad \text{Reflection Law} \tag{18}$$

$$\langle x, y \rangle \circ h = \langle x \circ h, y \circ h \rangle \qquad \text{Fusion Law} \tag{19}$$

$$(f \times g) \circ \langle x, y \rangle = \langle f \circ x, g \circ y \rangle \qquad \text{Functor Fusion Law} \tag{20}$$

The definition of product corresponds to the notion of product in **Set**

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

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 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

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

4.3.2 Coproducts

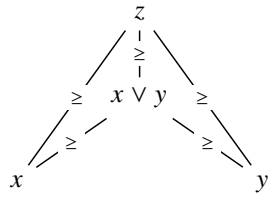
In logic, the *or* operator is introduced by stating that if Γ is a set of true propositions that entails A, or if Γ entails B, then $\Gamma \vdash A \vee B$. This is called the introduction rule of the coproduct:

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

The coproduct has one elimination rule given by:

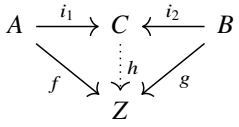
$$\frac{\Gamma \vdash A \vee B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}$$

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:



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 $i_1 : A \to C$ and $i_2 : B \to C$ called* injections *such that for any object Z with arrows $f : A \to Z$ and $g : B \to Z$, there exists a unique morphism $h : C \to Z$ making the following diagram commute:*



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

It should be clear by now that we can view the properties of this objects under the lenses of the homsets as well:

$$C(A + B, Z) \cong C \times C((A, B), (Z, Z))$$

This is derived by the unique mapping property of the coproduct:

$$f = [g_1, g_2] \iff f \circ \texttt{inl} = g_1 \text{ and } f \circ \texttt{inr} = g_2 \tag{21}$$

This yields the following properties:

$$[g_1, g_2] \circ \texttt{inl} = g_1 \text{ and } [g_1, g_2] \circ \texttt{inr} = g_2 \qquad \beta\text{-law} \tag{22}$$

$$h = [h \circ \texttt{inl}, h \circ \texttt{inr}] \qquad \eta\text{-law} \tag{23}$$

$$[\texttt{inl}, \texttt{inr}] = id_{A+B} \qquad \text{Reflection Law} \tag{24}$$

$$f \circ [g_1, g_2] = [f \circ g_1, f \circ g_2] \qquad \text{Fusion Law} \tag{25}$$

$$[g_1, g_2] \circ h_1 + h_2 = [g_1 \circ h_1, g_2 \circ h_2] \qquad \text{Functor Fusion Law} \tag{26}$$

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

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

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 $f : A \to Z$ and $g : B \to Z$, 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 $C \cong D$.*

4.3.3 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 (27) 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 $\epsilon : B^A \times A \to B$ *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 (27) commutes.*

$$\begin{array}{ccc} B^A & & B^A \times A \xrightarrow{\epsilon} B \\ {\scriptstyle \Lambda f}\uparrow\vdots & & {\scriptstyle \Lambda f \times A}\uparrow \quad \nearrow_{f} \\ Z & & Z \times A \end{array} \tag{27}$$

Notice, here Λ can be seen as the currying operation taking maps of type $Z \times A \to B$ to maps to type $Z \to B^A$. Conversely, if we happen to have a map $Z \xrightarrow{g} B^A$ then we can easily construct a map $Z \times A \xrightarrow{\forall g} B$ 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

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

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

This isomorphism is derived by the uniqueness property:

$$g = \Lambda f \iff \epsilon \circ g \times A = f \tag{28}$$

which yields the following additional properties:

$$\epsilon \circ \Lambda f \times A = f \qquad \beta\text{-law} \tag{29}$$
$$h = \Lambda(\epsilon \circ h \times A) \qquad \eta\text{-law} \tag{30}$$
$$\Lambda\epsilon = id_{B^A} \qquad \text{Reflection Law} \tag{31}$$
$$\Lambda f \circ g = \Lambda(f \circ g) \qquad \text{Fusion Law} \tag{32}$$
$$(g)^A \circ \Lambda f = \Lambda(g \circ f) \qquad \text{Functor Fusion Law} \tag{33}$$

4.4 Exercises

1. Prove the isomorphism
$$\mathtt{Str}A \cong \mathbf{Set}(\mathbb{N}, A)$$
is natural in A. Also construct the maps `head`, `tail` and `cons` for the type $\mathbb{N} \to A$

2. 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
$$A \longleftarrow P \longrightarrow B$$
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.

3. Let $\mathcal{D}$ be a locally small category, and let A, B, E be objects in $\mathcal{D}$, then there is an isomorphism
$$\mathcal{D}(X, E) \cong \mathcal{D}(X \times A, B)$$
natural in X if and only if E is an exponential object of B to the A (i.e. there exists an $\epsilon : E \times A \to B$ such that (E, ϵ) is an exponential).

   When using the 'only if' direction of the lemma, you should apply standard practice: construct the isomorphism, and then say the magic words: "it is obvious that this is natural in X".

4. Let C be a locally small category with finite products, let $F : C \to C$ and $G : \mathcal{D} \to C$ be a part of an equivalence of categories, and let A, B be the objects of $\mathcal{D}$. Then $F(1)$ is a terminal object in $\mathcal{D}$ and
$$A \times B \stackrel{\Delta}{=} F(GA \times GB)$$
defines a product of A and B.

5. Consider the category $\mathbf{Set}^{\mathrm{op}}$, the opposite category of **Set**, and the category of *complete atomic Boolean algebras* (**CABA**).

6. Define the objects and morphisms in $\mathbf{Set}^{\mathrm{op}}$ and **CABA**.

7. Show that there exists a *contravariant functor* $F : \mathbf{Set}^{\mathrm{op}} \to \mathbf{CABA}$ that:

   - Takes a set A to the Boolean algebra $\mathcal{P}(A)$, the power set of A.
   - Takes a function $f : A \to B$ in $\mathbf{Set}^{\mathrm{op}}$ and maps it to a *reversed* Boolean algebra homomorphism $h : \mathcal{P}(B) \to \mathcal{P}(A)$.

8. Prove that this functor establishes an *equivalence of categories* between $\mathbf{Set}^{\mathrm{op}}$ and **CABA**.

9. In **Set**, a subobject of the NNO is a set $P \subseteq \mathbb{N}$ with two maps $1 \to P$ and $P \to P$. Show that there is no proper subobject of the natural numbers in **Set**, in other words, prove that any subobject of $\mathbb{N}$ must be $\mathbb{N}$ itself.

10. Define a program $\mathtt{sum} : \mathbb{N} \to \mathbb{N}$ using the unique mapping property of the NNO which maps $n \in \mathbb{N}$ into the sum of all numbers from 0 to n

11. Using the properties of the NNO, prove that for all $n \in \mathbb{N}$, the sum of the first n natural numbers is given by the formula:

$$\sum_{k=0}^{n} k = \frac{n(n+1)}{2}.$$

12. 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.

13. Prove the following morphisms are isomorphisms by showing there for each of these there exists an inverse using the unique mapping property of the product:

$$\mathsf{unit}^{\times}_{A} : A \times 1 \to A \tag{34}$$
$$\mathsf{symm}^{\times}_{A,B} : A \times B \to B \times A \tag{35}$$
$$\mathsf{assoc}^{\times}_{A,B,C} : A \times (B \times C) \to (A \times B) \times C \tag{36}$$

14. Construct a map

$$\mathsf{copy}_{A} : A \to A \times A$$

    which makes a copy of the argument using the unique mapping property of the product

15. Given a map $f : X \to Y$ define the map $X \times A \to Y \times A$ such that it applies f to the left argument and leaves the right-hand side intact. Call this map $f \times A$. Similarly define $A \times f : A \times X \to A \times Y$. Finally, for a map $f : X \to Y$ and $g : X' \to Y'$ define a map

$$f \times g : X \times X' \to Y \times Y'$$

such that f is applied to the first component and g is applied to the second component

16. Prove that for all sets $X \in \mathbf{Set}$, the homset $\mathbf{Set}(1, X)$ is in bijective correspondence with X

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

18. 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 $!_B$

19. 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.

20. Let $\mathcal{C}$ be a cartesian closed category with an initial object 0. Show that for any object X in $\mathcal{C}$, $X \times 0$ is initial. Conclude $X \times 0 \cong 0$.

21. 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 $\mathcal{C}$ 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 1 and Lemma 2, which you do not have to prove. (They can be proved by induction on t and s respectively.)

5 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.

5.1 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

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

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

$$\begin{array}{rcll} t \in \text{Terms} & ::= & x & \text{(terms variables)} \\ & | & \texttt{zero} \mid \texttt{succ}\ (t) \mid \texttt{fold}\ t_1\ t_2\ t_3 & \text{(natural numbers)} \\ & | & (t_1, t_2) \mid \texttt{prj}_1(t) \mid \texttt{prj}_2(t) & \text{(products)} \\ & | & \text{case } t \text{ of } \{\texttt{inl}(x) \Rightarrow t_1; \texttt{inr}(y) \Rightarrow t_2\} \mid \texttt{inl}(t) \mid \texttt{inr}(t) & \text{(coproducts)} \\ & | & \lambda x.t \mid t_1 t_2 & \text{(functions)} \end{array}$$

Equational Laws

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

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{}{\Gamma \vdash () : \mathsf{unit}} \qquad \frac{}{\Gamma \vdash \texttt{zero} : \mathsf{nat}} \qquad \frac{\Gamma \vdash t : \texttt{nat}}{\Gamma \vdash \texttt{succ}\ (t) : \texttt{nat}}$$

$$\frac{\Gamma \vdash t_1 : \texttt{nat} \qquad \Gamma \vdash t_2 : A \qquad \Gamma \vdash t_3 : A \to A}{\Gamma \vdash \texttt{fold}\ t_1\ t_2\ t_3 : A}$$

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

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

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

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

5.2 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:

$$\begin{aligned}\llbracket . \rrbracket &= 1\\ \llbracket \Gamma, x : A \rrbracket &= \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket\end{aligned}$$

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

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

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

And, finally, the function $\llbracket \cdot \rrbracket : \text{Terms} \to \text{Arr}(C)$ 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 $\llbracket t \rrbracket$ of type $\llbracket \Gamma \rrbracket \to \llbracket A \rrbracket$, therefore abusing some notation the correct type of the interpretation would be something like:

$$\llbracket \Gamma \vdash t : A \rrbracket : \llbracket \Gamma \rrbracket \xrightarrow{\llbracket t \rrbracket} \llbracket A \rrbracket$$

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

$$\llbracket \Gamma, x : A \vdash t : B \rrbracket : \llbracket \Gamma \rrbracket \times \llbracket A \rrbracket \to \llbracket B \rrbracket$$

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

$$\begin{aligned}
\llbracket \Gamma \vdash () : \mathtt{unit} \rrbracket &= !\\
\llbracket \Gamma \vdash x : A \rrbracket &= \pi_x\\
\llbracket \Gamma \vdash \mathtt{zero} : \mathtt{nat} \rrbracket &= \mathsf{zero} \circ !\\
\llbracket \Gamma \vdash \mathtt{succ}(t) : \mathtt{nat} \rrbracket &= \mathsf{succ} \circ \llbracket t \rrbracket\\
\llbracket \Gamma \vdash \mathtt{fold}\ t_1\ t_2\ t_3 : A \rrbracket &= \langle\!\langle\, [\Lambda \llbracket t_2 \rrbracket, \Lambda(\forall \llbracket t_3 \rrbracket \circ \mathsf{symm}_\times \circ \epsilon \times \mathsf{copy})] \,\rangle\!\rangle \circ \llbracket t_1 \rrbracket\\
\llbracket \Gamma \vdash \mathtt{prj}_1(t) : A \rrbracket &= \pi_1 \circ \llbracket t \rrbracket\\
\llbracket \Gamma \vdash \mathtt{prj}_2(t) : B \rrbracket &= \pi_2 \circ \llbracket t \rrbracket\\
\llbracket \Gamma \vdash (t_1, t_2) : A \times B \rrbracket &= \langle \llbracket t_1 \rrbracket, \llbracket t_2 \rrbracket \rangle\\
\llbracket \Gamma \vdash \mathtt{inl}(t) : A \rrbracket &= i_1 \circ \llbracket t \rrbracket\\
\llbracket \Gamma \vdash \mathtt{inr}(t) : A \rrbracket &= i_2 \circ \llbracket t \rrbracket\\
\llbracket \Gamma \vdash \text{case } t \text{ of } \{\mathtt{inl}(x) \Rightarrow t_1; \mathtt{inr}(y) \Rightarrow t_2\} : A \rrbracket &= [\Lambda(\llbracket t_1 \rrbracket \circ \mathsf{symm}_\times), \Lambda(\llbracket t_2 \rrbracket \circ \mathsf{symm}_\times)] \circ \llbracket t \rrbracket\\
\llbracket \Gamma \vdash \lambda x.M : A \to B \rrbracket &= \Lambda \llbracket M \rrbracket\\
\llbracket \Gamma \vdash MN : B \rrbracket &= \epsilon \circ \langle \llbracket M \rrbracket, \llbracket N \rrbracket \rangle
\end{aligned}$$

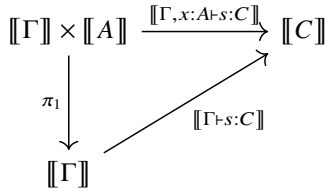
where $! : \llbracket \Gamma \rrbracket \to 1$ is the unique map into the terminal object and $n : \Gamma \to \mathbb{N}$ is the map disregarding the context and returning the number denoted syntactically by $\underline{n}$. In **Set**, for example, there would be a constant map $\lambda\gamma.n$ for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

5.3 Exercises

1. 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

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

2. If $\Gamma \vdash s : C$ is a valid typing judgment, so is $\Gamma, x : A \vdash s : C$ and the following diagram is commutative



6 Limits and Colimits

In category theory, a *product* is a specific example of a *limit*, capturing the idea of combining objects in a universal way. Given two objects A and B in a category C, their product $A \times B$ is defined as the limit of a diagram consisting of A and B with no morphisms between them.

In a sense the two objects are the shape or diagram, while the product is the best object that has two arrows going into A and B. This is the intuition that leads to the definition of categorical limit.

6.1 Limits

We first define formally the notion of *shape* or *diagram*.

Definition 6.1 (Diagrams)**.** *Given an index category $\mathcal{I}$ and a category C, a* diagram *is a functor $D : \mathcal{I} \to C$.*

We would like now to consider all the objects which have projections into all the objects given by the diagram, that is all objects $\{D_i\}_{i \in \mathcal{I}}$. For a diagram $D : \mathcal{I} \to C$ we call these the cones for D.

Definition 6.2 (Cone)**.** *For a diagram $D : \mathcal{I} \to \mathcal{D}$, a* cone *is an object C such that there exists a family of maps $c_i : C \to D_i$ and such that*

$$D(f) \circ c_i = c_j \tag{37}$$

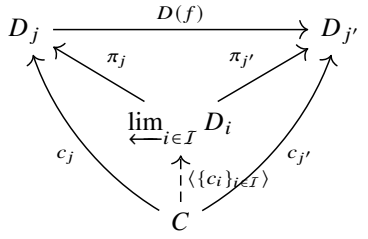
for every $f : i \to j$.

Note that this definition has an additional condition stating that the projections must respect the arrows between objects. We now would like to take the *best* among these cones. This particular object is called the *limit* of a diagram D.

Definition 6.3 (Limit)**.** *For a diagram $D : \mathcal{I} \to \mathcal{C}$, a* limit *is a universal cone P, that is a cone that for any other cone $\{c_i : C \to D_i\}_{i\in\mathcal{I}}$ there exists a unique arrow $\langle\{c_i\}_{i\in\mathcal{I}}\rangle : C \to \varprojlim_{i\in\mathcal{I}} D_i$ such that*

$$\pi_i \circ \langle\{c_i\}_{i\in\mathcal{I}}\rangle = c_i \tag{38}$$

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



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

Similarly to what we have done throughout this section the following isomorphism hold:

$$\mathcal{C}^{\mathcal{I}}(\Delta X, D) \cong \mathcal{C}(X, \varprojlim_{i\in\mathcal{I}} D_i)$$

where $\Delta X = (X, X)$ is the diagonal functor with the obvious functorial action. This states that given a natural transformation between ΔX and D, that is a family of maps $\{X \to D_i\}_{i\in\mathcal{I}}$ there exists only one map $X \to \varprojlim_{i\in\mathcal{I}} D_i$ making the above isomorphism natural in X and D. This is derived by the unique mapping property of the coproduct:

$$g = \langle\{c_i\}_{i\in\mathcal{I}}\rangle \iff \pi_i \circ g = c_i \tag{39}$$

This yields the following properties:

$$\pi_j \circ \langle\{c_i\}_{i\in\mathcal{I}}\rangle = c_j \qquad \beta\text{-law} \tag{40}$$

$$h = \langle\{\pi_i \circ h\}_{i\in\mathcal{I}}\rangle \qquad \eta\text{-law} \tag{41}$$

$$\langle\{\pi_i\}_{i\in\mathcal{I}}\rangle = id_{\varprojlim_{i\in\mathcal{I}} D_i} \qquad \text{Reflection Law} \tag{42}$$

$$\langle\{c_i\}_{i\in\mathcal{I}}\rangle \circ g = \langle\{c_i \circ g\}_{i\in\mathcal{I}}\rangle \qquad \text{Fusion Law} \tag{43}$$

$$\varprojlim h \circ \langle\{c_i\}_{i\in\mathcal{I}}\rangle = \circ\langle\{h \circ c_i\}_{i\in\mathcal{I}}\rangle \qquad \text{Functor Fusion Law} \tag{44}$$

If we now take the limit of all the maps $X \to D_i$ then this is isomorphic to the maps into the limit of D. This corresponds to saying that the homset *functor preserves limits*

$$\lim_{i\in\mathcal{I}} \mathcal{C}(X_i, D_i) \cong \mathcal{C}(X, \varprojlim_{i\in\mathcal{I}} D_i) \tag{45}$$

More in general, a functor $F : \mathcal{C} \to \mathcal{D}$ is *continuous* when it preserves limits

Definition 6.4 (Continuous Functors)**.** *A functor $F : \mathcal{C} \to \mathcal{D}$ is called continuous if it preserves limits, that is the following isomorphism holds:*

$$F(\varprojlim_{i\in\mathcal{I}} D_i) \cong \varprojlim_{i\in\mathcal{I}} (F \circ D_i)$$

A myriad of definitions stem from the one of the limit. Given the category **2** with only two objects and no maps between them, the limit of a diagram $D : \mathbf{2} \to C$ is the product.

In general, a diagram D is a family of objects $\{A_i\}_{i \in \mathcal{I}}$ indexed by an *index category* $\mathcal{I}$. When there are no arrows in $\mathcal{I}$ we call this category a *discrete category* . Then the *best object* which has a projection into all of these objects $\{A_i\}_{i \in \underline{n}}$ is called the *dependent product*, denoted by $\Pi_{i \in \mathcal{I}} A_i$.

The isomorphisms work similarly to the case of the binary product above, that is, for each family of projection maps $f_i : X \to A_i$ there exists only one map $X \to \Pi_{i \in \underline{n}} A_i$, that is the pairing of all the generalised elements f_i

$$C^{\mathcal{I}}(\Delta X, D) \cong C(X, \Pi_{i \in \mathcal{I}} A_i)$$

Because the representable homset functor preserves limits (45) specialises as follows:

$$\Pi_{i \in \mathcal{I}} C(X, A_i) \cong C(X, \Pi_{i \in \mathcal{I}} A_i)$$

When the diagram functor is constant, or when the family of objects is the same constant object for all $i \in \mathcal{I}$ the dependent product specialises to $\Pi_{i \in \mathcal{I}} A$ which is called the *power*. When $\mathcal{I}$ is small and A is a set, the power $A^{\mathcal{I}}$ is the exponential object in **Set** which corresponds to the function space $\mathcal{I} \to A$. Notice that $\mathcal{I}$ is discrete, therefore also $\mathcal{I}$ can be seen as a set.

Again (45) specialises to

$$\mathcal{I} \to C(X, A) \cong (X, A^{\mathcal{I}})$$

6.2 Colimits

Dually we define *cocones* and *colimits*.

Definition 6.5 (Cocone)**.** *For a diagram $D : \mathcal{I} \to \mathcal{D}$, a* cocone *is an object C such that there exists a family of maps $c_i : D_i \to C$ and such that*

$$c_j \circ D(f) = c_i \tag{46}$$

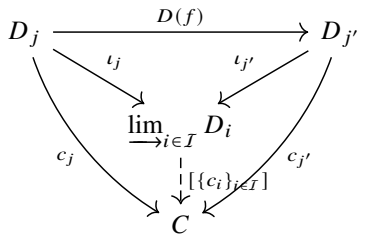
for every $f : i \to j$.

Note that this definition includes a coherence condition stating that the injections must respect the arrows between objects. We now wish to take the *best* among these cocones. This particular object is called the *colimit* of a diagram D.

Definition 6.6 (Colimit)**.** *For a diagram $D : \mathcal{I} \to C$, a* colimit *is a universal cocone P, that is, a cocone such that for any other cocone $\{c_i : D_i \to C\}_{i \in \mathcal{I}}$, there exists a unique arrow $[\{c_i\}_{i \in \mathcal{I}}] : \varinjlim_{i \in \mathcal{I}} D_i \to C$ such that*

$$[\{c_i\}_{i \in \mathcal{I}}] \circ \iota_i = c_i \tag{47}$$

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



We denote the colimit of a diagram D by $\varinjlim_{i\in\mathcal{I}} D_i$.

Similarly, the following natural isomorphism holds:

$$C(\varinjlim_{i\in\mathcal{I}} D, X) \cong C^{\mathcal{I}}(D, \Delta X)$$

where $\Delta X = (X, X)$ is the diagonal functor with the obvious functorial action. This states that given a natural transformation between D and ΔX, that is a family of maps $\{D_i \to X\}_{i\in\mathcal{I}}$, there exists only one map $\varinjlim_{i\in\mathcal{I}} D_i \to X$ making the above isomorphism natural in X and D.

This is derived by the unique mapping property of the colimit:

$$g = [\{c_i\}_{i\in\mathcal{I}}] \iff g \circ \iota_i = c_i \tag{48}$$

This yields the following properties:

$$[\{c_i\}_{i\in\mathcal{I}}] \circ \iota_j = c_j \qquad \beta\text{-law} \tag{49}$$

$$h = [\{h \circ \iota_i\}_{i\in\mathcal{I}}] \qquad \eta\text{-law} \tag{50}$$

$$[\{\iota_i\}_{i\in\mathcal{I}}] = id_{\varinjlim_{i\in\mathcal{I}} D_i} \qquad \text{Reflection Law} \tag{51}$$

$$g \circ [\{c_i\}_{i\in\mathcal{I}}] = [\{g \circ c_i\}_{i\in\mathcal{I}}] \qquad \text{Fusion Law} \tag{52}$$

$$[\{c_i\}_{i\in\mathcal{I}}] \circ \varinjlim h = [\{c_i \circ h\}_{i\in\mathcal{I}}] \qquad \text{Functor Fusion Law} \tag{53}$$

More generally, a functor $F : \mathcal{C} \to \mathcal{D}$ is *cocontinuous* when it preserves colimits.

Definition 6.7 (Cocontinuous Functors)**.** *A functor $F : \mathcal{C} \to \mathcal{D}$ is called cocontinuous if it preserves colimits, that is the following isomorphism holds:*

$$F(\varinjlim_{i\in\mathcal{I}} D_i) \cong \varinjlim_{i\in\mathcal{I}} (F \circ D_i)$$

However, the representable homset functor does not preserve colimits on its contravariant argument, rather it turns colimits into limits, it *reverses colimits*:

$$C(\varinjlim_{i\in\mathcal{I}} A_i, X) \cong \varprojlim_{i\in\mathcal{I}} C(A_i, X)$$

Similarly to what we have done in the case of limits we can now consider what happens in the case where the index category is discrete. In this case the colimit specialises to the *dependent sum* $\sum_{i \in \mathcal{I}} A_i$.

$$C(\Sigma_{i \in \mathcal{I}} A_i, X) \cong C^{\mathcal{I}}(A, \Delta X)$$

Moreover, if the diagram $A : \mathcal{I} \to C$ is constant then we write $\sum \mathcal{I}.A$, which is called the *copower*, also written $\mathcal{I} \bullet A$. In **Set**, when $\mathcal{I}$ is small this specialises to the product $\mathcal{I} \times A$.

$$C(\Sigma \mathcal{I} A, X) \cong C^{\mathcal{I}}(A, \Delta X)$$

Now recall that hom-functors preserve limits and reverse colimits:

$$C(\Sigma \mathcal{I}.A, X) \cong \Pi \mathcal{I}.C(A, X) \tag{54}$$

$$\Pi \mathcal{I}.C(X, B) \cong C(X, \Pi \mathcal{I}.B) \tag{55}$$

Since these isomorphisms are natural in X we can specialise (54) with B and (55) with A and derive

$$C(\Sigma \mathcal{I}.A, X) \cong \Pi \mathcal{I}.C(A, B) \cong C(A, \Pi \mathcal{I}.B)$$

This is in essence the isomorphism given by the currying and uncurrying operations for exponetials.

6.3 The Yoneda Lemma

Consider a typed language with a unary constructor $\mathtt{R}$ that follows the typing rule

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash \mathtt{R}(t) : B}. \tag{56}$$

Our goal is to provide a semantic interpretation $[\![\cdot]\!]$ for this language by induction on the typing judgment $\Gamma \vdash t : A$, such that terms are mapped to morphisms $[\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![A]\!]$, assuming $[\![\cdot]\!]$ is also separately defined for contexts and types.

To interpret (56), we apply induction on the typing judgment. By assumption, we already have a morphism $[\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![A]\!]$, and our task is to construct a morphism $[\![\Gamma]\!] \xrightarrow{[\![\mathtt{R}(t)]\!]} [\![B]\!]$.

For simplicity, in the remainder of this section, we omit semantic brackets where clear from context. For example, we assume A represents $[\![A]\!]$, and similarly, $t : \Gamma \to A$ represents the interpretation of t.

Determining the correct morphism in this situation can be nontrivial due to the need to manage context correctly. However, a particular instantiation of the Yoneda lemma provides a canonical way to construct the desired morphism. Specifically, given a morphism $t : \Gamma \to A$ and a morphism $R : A \to B$, the Yoneda lemma ensures the existence of a canonical morphism $\Gamma \to B$ which is given by the following isomorphism:

$$C(A, B) \cong \mathrm{Nat}(C(-, A), C(-, B))$$

Let $R : A \to B$ be the interpretation of R. By the isomorphism, the corresponding natural transformation satisfies $\phi(R, t) = F(t)(R) = C(t, B)(R)$, which simply reduces to $R \circ t$. Thus, the interpretation of $R(t)$ is given by composition: $R \circ t$.

The Yoneda Lemma is a fundamental result in category theory that describes how functors of type $C^{\mathsf{op}} \to \mathbf{Set}$, called *presheaves*, interact with natural transformations. More specifically, this result tells us that giving a natural transformation $\eta : C(-, A) \to F$ is equivalent to specifying an element of $F(A)$. In other words, the presheaf F is completely determined by how it interacts with the representable functor $C(-, A)$.

Lemma 6.1 (Yoneda)**.** *Given a locally small category C and a functor $F : C^{\mathsf{op}} \to \mathbf{Set}$, the Yoneda Lemma states that for any object A in C, there is a natural isomorphism*

$$F(A) \cong Nat(C(-, A), F)$$

where $Nat(C(-, A), F)$ denotes the set of natural transformations from the hom-functor $C(-, A)$ to F, and $F(A)$ is the value of F at A.

Sketch. To see why this holds, we construct a correspondence between elements of $F(A)$ and natural transformations $C(-, A) \to F$. Given a natural transformation $\phi : C(-, A) \to F$, we obtain an element of $F(A)$ by evaluating ϕ_A at the identity morphism, that is, setting $x = \phi_A(\mathrm{id}_A) \in F(A)$. Conversely, given an element $x \in F(A)$, we define a natural transformation ϕ by setting $\phi_B(f) = F(f)(x)$ for any morphism $f : B \to A$. The fact that these two operations are inverse to each other establishes the natural isomorphism. □

6.3.1 The Yoneda Embedding

The representable homset functor $C(-, =) : C^{\mathsf{op}} \times C \to \mathbf{Set}$ can be rewritten as a functor of type $C \to \mathbf{Set}^{C^{\mathsf{op}}}$ defined as $A \mapsto C(-, A)$. This functor is called the Yoneda embedding and we denote it by $\yoneda$. Whenever $A : C$ we write $\yoneda_A$ to mean $C(-, A)$. The Yoneda embedding is called so because it is fully-faithful, i.e. it preserves and reflects isomorphisms of objects, and it is a bijection on arrows. This implies directly the following principle.

Proposition 6.1 (The Yoneda Principle)**.** *The Yoneda embedding*

- *For all objects $A, B : C$*

$$A \cong B \iff \yoneda_A \cong \yoneda_B$$

- *For all morphisms $f, g : A \to B$*

$$f = g \iff \yoneda_f = \yoneda_g$$

Note that the functorial action of $\yoneda$, namely, $C(-, f)$ for a morphism $f : A \to B$, has the following type

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

taking a morphism to the natural transformations between the contravariant presheaves $C(-, A)$ and $C(-, B)$. By direct application of the Yoneda lemma with F being $(-, B)$ this is an isomorphism thus proving $よ$ is fully faithful.

An immediate application of the Yoneda lemma is proving isomorphisms of objects. Say for example we want to prove

$$A \times (B \times C) \cong (A \times B) \times C$$

we prove instead that

$$よ_{(A\times B)\times C} \cong よ_{A\times(B\times C)}$$

Because $よ$ is a representable functor it preserves limits

$$よ_{(A\times B)\times C} \cong よ_A \times (よ_B \times よ_C)$$

and now the product on the right is a product in $\mathbf{Set}^{C^{\mathsf{op}}}$ which is defined point-wise as

$$(F \times G)(C) = FC \times GC$$

and similarly for the arrows. Thus it boils down to a product in **Set** which is obviously associative. Hence we can derive

$$よ_A \times (よ_B \times よ_C) \cong (よ_A \times よ_B) \times よ_C$$

we proves associativity of the product without having to use the unique mapping property of the product in C as in Exercise 13.

we can instead prove that $C(A \times (B \times C), -) \cong C((A \times B) \times C, -)$. Note that we are not using exactly $よ$, but the Yoneda principle works as a fine with the covariant case. Since the isomorphism is natural in the first argument of the homset functor we assume $X : C$ then we compute:

$$\begin{aligned} C(A \times (B \times C), X) &\cong C(B \times C, X^A) \\ &\cong C(B, (X^A)^C) \\ &\cong C(B, (X^C)^A) \\ &\cong C(B \times A, X^C) \\ &\cong C(A \times B, X^C) \\ &\cong C((A \times B) \times C, X) \end{aligned}$$

Note that we used the fact that $(X^A)^C \cong (X^C)^A$. This is also easy to prove using the Yoneda principle in the same way.

6.4 Exercises

1. The notion of colimit is obtained by reversing the arrows. Derive it by exercise.

7 Data Types as Fixed-Points

7.1 Least Fixed-Points as Colimits

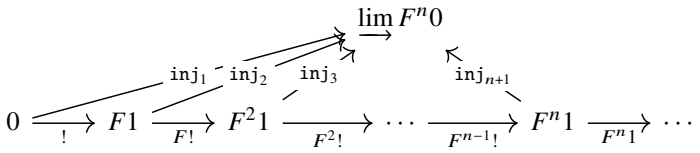
Let $F : C \to C$ be an endofunctor and let ω be the first infinite ordinal, i.e. the set of natural numbers with the canonical ordering. When ω is viewed as a category it has the following shape:

$$0 \longrightarrow 1 \longrightarrow 2 \longrightarrow \cdots \longrightarrow n \longrightarrow \cdots$$

Let $D : \omega \to C$ be a diagram defined on objects $D(0) = 0$ and $D(n+1) = FD(n)$ and arrows $D(0 \leq 1) = 0 \xrightarrow{!} F0$ and $D(n \leq (n+1)) = F^n 0 \xrightarrow{FD(n-1 \leq n)} F^{(n+1)}0$. We can view D as the following diagram

$$0 \xrightarrow{!} F0 \xrightarrow{F!} F^2 0 \longrightarrow \cdots \longrightarrow F^n 0 \xrightarrow{F^n !} \cdots$$

We now take the colimit of this diagram



and prove that if F is cocontinuous then it as a least fixed-point. A *fixed-point* for a functor F is an object X such that $FX \cong X$, we write $\mathrm{In} : FX \to X$ for the isomorphism. A *least fixed-point* for F is a fixed-point, denoted by μF, such that for any other fixed-point $FY \cong Y$, with β witnessing the isomorphism, there exists a unique mapping $(\!|\beta|\!) : \mu F \to Y$ such that $(\!|\beta|\!) \circ \mathrm{In} = \beta \circ F(\!|\beta|\!)$, that is the following diagram commutes:

$$\begin{array}{ccc} F(\mu F) & \xrightarrow{F(\!|\beta|\!)} & FY \\ {\scriptstyle \mathrm{In}}\downarrow & & \downarrow{\scriptstyle \beta} \\ \mu F & \xrightarrow{(\!|\beta|\!)} & Y \end{array}$$

The choice of notation for the unique mapping out of the least fixed-point will explained shortly. We first prove that this colimit gives a fixed-point. We compute as follows using the fact that F is continuous (Definition 6.7) and that $\varinjlim_{n \in \omega} F^n 0 \cong \varinjlim_{n \in \omega} F^{n+1} 0$

which is to be verified separately as an exercise for the reader:

$$\begin{aligned}
\mu F &= \varinjlim_{n\in\mathbb{N}} F^n 0 \\
&\cong \varinjlim_{n\in\mathbb{N}} F^{n+1} 0 \\
&\cong \varinjlim_{n\in\mathbb{N}} F F^n 0 \\
&\cong F\left(\varinjlim_{n\in\mathbb{N}} F^n 0\right) \\
&\cong F(\mu F)
\end{aligned}$$

We now prove that this is the least fixed-point. So assume there is another fixed-point $\beta : FY \cong Y$. We construct $(\!|\beta|\!)$ by using the unique mapping property of the coproduct (Definition 4.2). We have to show that Y is a cocone for the ω-chain $\{F^n 0\}_{n\in\omega}$. By induction on the natural numbers, the base case is $0 \xrightarrow{!} Y$. Assuming $F^n 0 \to Y$ we can use the functorial action of F to get $F^{n+1}0 \to FY$ and post-compose it with isomorphism β. Checking that the coherence conditions work is an easy exercise. Now since Y is a cocone there exists a unique map $\mu F \to Y$ which is what we wanted.

Exercise 7.1. *Let $FX = 1 + X$. Prove the colimit of the ω-chain $\{F^n 0\}$ is the natural numbers object. Hint: prove that F is cocontinuous, construct the least fixed-point of F. Once you have the isomorphism, construct the maps* zero *and* succ *and prove the universal property of the NNO using the universal property of the colimits.*

7.2 Greatest Fixed-Points as Limits

We now dualize the construction of least fixed-points via colimits to obtain *greatest fixed-points* using *limits*. As before, let $F : C \to C$ be an endofunctor. Previously, we used the category ω of natural numbers with their standard ordering to build a diagram from the initial object upwards via repeated application of F. For the dual construction, we instead consider the opposite category ω^{op}, and build a diagram that goes downward from the terminal object.

Define a diagram $D : \omega^{\mathrm{op}} \to C$ as follows: let $D(0) = 1$, the terminal object, and define $D(n+1) = F(D(n))$. The morphisms $D(n+1 \to n)$ are given by $F^n! : F^{n+1}1 \to F^n 1$, by applying F to the unique map $! : 1 \to 1$. This yields a diagram of the form

$$\cdots \to F^n 1 \to \cdots \to F^2 1 \xrightarrow{F!} F1 \xrightarrow{!} 1$$

We then define the greatest fixed-point as the limit of this diagram:

$$\nu F := \varprojlim_{n\in\omega^{\mathrm{op}}} F^n 1$$

Assuming F is *continuous*, i.e., it preserves limits of ω-chains, we compute:

$$\begin{aligned}
\nu F &= \varprojlim_{n \in \omega^{\mathrm{op}}} F^n 1 \\
&\cong \varprojlim_{n \in \omega^{\mathrm{op}}} F^{n+1} 1 \\
&\cong \varprojlim_{n \in \omega^{\mathrm{op}}} F(F^n 1) \\
&\cong F\left(\varprojlim_{n \in \omega^{\mathrm{op}}} F^n 1 \right) \\
&= F(\nu F)
\end{aligned}$$

so $\nu F \cong F(\nu F)$, and we have a fixed-point. We denote this isomorphism by $\mathrm{Out} : \nu F \to F(\nu F)$, in line with the convention for coalgebras.

To show that this is the *greatest* fixed-point, suppose we are given another fixed-point $\beta : Y \xrightarrow{\cong} FY$. We aim to construct a unique map $[\![\beta]\!] : Y \to \nu F$ such that the following diagram commutes:

$$\begin{array}{ccc}
Y & \xrightarrow{[\![\beta]\!]} & \nu F \\
{\scriptstyle \beta}\downarrow & & \downarrow{\scriptstyle \mathrm{Out}} \\
FY & \xrightarrow[F[\![\beta]\!]]{} & F(\nu F)
\end{array}$$

This map $[\![\beta]\!]$ arises from the universal property of the limit. Since $Y \cong FY$, we can inductively define maps $Y \to F^n 1$ for each n, beginning with the unique map $Y \to 1$, and using β^{-1} to define a map from Y into $F^{n+1}1$ whenever we have one into $F^n 1$. Thus, Y forms a cone over the diagram $\{F^n 1\}$, and the universal property of the limit gives the unique map $[\![\beta]\!] : Y \to \nu F$ making the square commute.

Exercise 7.2. *Let $FX = A \times X$. Prove the limit of the ω^{op}-chain $\{F^n 1\}$ is the type of streams over A. Hint: prove that F is continuous, construct the greatest fixed-point of F. Once you have the isomorphism, construct the maps* head *and* tail.

8 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.

8.1 Adjunctions

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

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

[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.

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

$$C \underset{R}{\overset{L}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathcal{D}$$

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 $L \dashv R$.

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 $g : A \to RB$

$$R(b) \cdot \lfloor f \rfloor = \lfloor b \cdot f \rfloor \tag{57}$$
$$\lfloor f \rfloor \cdot a = \lfloor f \cdot L(a) \rfloor \tag{58}$$
$$b \cdot \lceil g \rceil = \lceil R(b) \cdot g \rceil \tag{59}$$
$$\lceil g \rceil \cdot L(a) = \lceil g \cdot a \rceil \tag{60}$$

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.

8.2 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

$$C \underset{\Delta}{\overset{0}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{1} \underset{1}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

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:

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

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

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

Exercise 8.1. *Compute the two naturality conditions and derive the fusion laws.*

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

8.3 (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:

$$C \underset{\Delta}{\overset{+}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C \times C \underset{\times}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

Exercise 8.2. *Derive the fusion laws and conclude these are exactly those of the product and coproduct respectively.*

8.4 Exponentials

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

$$C \underset{(-)^A}{\overset{(-)\times A}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

The isomorphism induced by this adjunction is stated as follows:

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

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

8.5 (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:

$$C \underset{\Delta}{\overset{\varinjlim}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C^I \underset{\varprojlim}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} C$$

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.

9 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 [10] 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:

$$\forall\alpha.\mathsf{List}\ \alpha \to \mathsf{List}\ \alpha \tag{61}$$

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 (61) 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 [8] as a programming language known was *System F* or *second-order λ-calculus*.

For instance, the `reverse` function might operate on elements of type $\mathbb{N}$ or η, but also on the type (61) 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 5. 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 [9].

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

Models of impredicative polymorphism have eventually been found by Pitts [7] 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*).

9.1 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:

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

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:

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

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.

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

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:

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

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

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

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

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

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:

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

9.2 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.

$$\mathbf{Set}^{\mathcal{U}^n \mathcal{U}} \underset{\Pi}{\overset{\Delta}{\underset{\longrightarrow}{\overset{\longleftarrow}{\bot}}}} \mathbf{Set}^{\mathcal{U}^n}$$

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:

$$\lfloor \cdot \rfloor : \mathbf{Set}^{\mathcal{U}^{n+1}}(\Delta\Gamma, \tau) \cong \mathbf{Set}^{\mathcal{U}^n}(\Gamma, \Pi\tau) : \lceil \cdot \rceil \tag{62}$$

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 $[\![\Gamma]\!]$ is a object in $\mathbf{Set}^{\mathcal{U}^m}$:

$$[\![\Gamma]\!] = [\![T_1]\!] \times \cdots \times [\![T_n]\!]$$

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 $[\![\Gamma \vdash t : \tau]\!] : [\![\Gamma]\!] \xrightarrow{[\![t]\!]} [\![\tau]\!]$ in $\mathbf{Set}^{\mathcal{U}^n}$ whereas a well-typed term $\Gamma \vdash t : A$ is an arrow $[\![\Gamma \vdash t : A]\!] : [\![\Gamma]\!] \xrightarrow{[\![t]\!]}$

$[\![A]\!]$. The interpretation is then given by induction on the typing judgment:

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

Most of the terms are interpreted as in Section 5. Assuming a map $[\![t]\!] : \Delta[\![\Gamma]\!] \to [\![\tau]\!]$ 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 (62) defined above which is an arrow of type

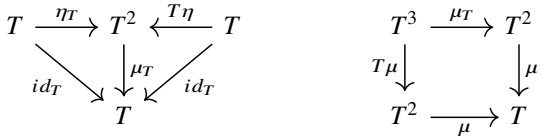
$$[\![\Gamma]\!]_\iota \to \Pi_{X \in \mathcal{U}} [\![\tau]\!]_{\iota[\alpha \mapsto X]}$$

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

10 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 10.1 (Monad)**.** *For a category C, a* monad *is an endofunctor $T : C \to C$ such that there exists two natural transformations $\eta : Id \dot{\to} T$ and $\mu : TT \dot{\to} T$ such that the following diagrams hold:*



Example 10.1 (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 10.2 (State Monad)**.** *Assume a set of state S and let* $T : \mathbf{Set} \to \mathbf{Set}$ *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 $\mu_A \circ T(f)$.

10.1 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.

10.2 The Kleisli Category

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

Definition 10.2 (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:

$$A \xrightarrow{f} TB \xrightarrow{T(g)} T^2C \xrightarrow{\mu_C} TC, \quad A \xrightarrow{(g \circ f)_T} TC$$

11 The Computational λ-calculus

The computational λ-calculus which we denote by λ_C is a calculus with effects first devised by Eugenio Moggi [6] 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 C_T for a (strong) monad T.

11.1 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:

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

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

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

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.

11.2 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 $[\![\,]\!] : \text{Types} \to \text{Obj}(C_T)$. Notice that $\text{Obj}(C_T)$ is exactly $\text{Obj}(C)$.

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

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 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 C

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

In order to explain the interpretation we work in the category 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:

$$\begin{array}{ccccc}
\Gamma & & \xrightarrow{\qquad\qquad\qquad\qquad\qquad\qquad} & & TB \\
& \searrow^{\langle t_1, t_2 \rangle} & \xrightarrow{\mathrm{app}(t_1,t_2)} & & \uparrow{\scriptstyle \mu_B} \\
T(TB^A) \times TA & & & & T^2B \\
& \searrow^{st_{TB^A,TA}} & & & \uparrow{\scriptstyle \mu_{TB}} \\
T(TB^A \times TA) & & \xrightarrow{T(st_{TB^A,A})} & T(T(TB^A \times A)) & \xrightarrow{T^2(\epsilon)} T^3B
\end{array}$$

Exercise 11.1. *Let T be the free monoids monad in* **Set***, i.e. the monad that maps a set A to the set of finite lists of elements of A. Show that the category of algebras for T is isomorphic to the category of monoids.*

Index

2-category, 12

ad-hoc polymorphism, 39
adjunction, 36
and, 17
antisymmetry, 5
arrow, 3
arrows, 7

best, 27, 29
best object, 29
bijection, 5

capture-avoiding substitution, 24
cardinality, 5
cartesian product, 4, 12
category, 8
category of categories, 11
category of functors, 11
cocone, 29
cocones, 29
cocontinuous, 30
colimit, 29
colimits, 29
complete atomic Boolean algebras, 23
cone, 27
continuous, 28, 36
contravariant, 13
contravariant functor, 23
contravariant hom-functor, 13
copower, 31
coproduct, 19
countable, 5
covariant hom-functor, 12

denotational semantics, 2
dependent product, 5, 29
dependent sum, 31
diagram, 27
discrete category, 29
disjoint union, 4

elements, 4
empty set, 4
equivalence of categories, 13, 23
equivalence relation, 4
equivalent, 13
evaluation map, 21
exponential, 21

faithful, 11
finite, 5
finite products, 25
fixed-point, 34
full, 11
function, 2, 4, 5
functor, 9, 10
functor preserves limits, 28
functorial action, 9, 10
functors, 12
fusion laws, 37

greatest, 36
greatest fixed-points, 35
greatest lower bound, 17

Heyting algebra, 21
homset functor, 11

impredicative, 39
index category, 29
indexed family of sets, 5
inductive proof principle, 16
infinite, 5
infinite product, 5
injections, 19
injective, 5
internal, 21
isomorphic, 5
isomorphisms of categories, 12

join, 19

least fixed-point, 34
least upper bound, 19
left adjoint, 37
limit, 27, 28
limits, 35

locally small, 8
lower semilattice, 17

meaning, 25
meet, 17
monad, 43
monotone function, 9
monotypes, 40

natural isomorphism, 11
natural numbers, 4
natural numbers object, 15
natural numbers with zero, 4
natural transformation, 11
natural transformations, 12
naturally isomorphic, 13

object, 2
objects, 7
opposite category, 12
or, 19

parametric, 39
partial order, 5
polytypes, 40
post-composition, 13
power, 29
pre-composition, 13
Predicative polymorphism, 39
predicative polymorphism, 40
preorder, 3, 5
presheaves, 32
product, 18, 27
product category, 12
program, 2

quotient, 4

reflexivity, 4
relation, 4
representable functors, 13
reversed, 23
reverses colimits, 30

second-order λ-calculus, 39
semantic brackets, 25
semantic function, 25
set, 2, 4
set containing all sets, 39
shape, 27
singleton set, 4
size, 5
small, 8, 41
small categories, 11
state monad, 44
structure-preserving, 9
surjective, 5
symmetry, 4
System F, 39

terminal, 15
transitivity, 5
type, 2
type schemes, 40

union, 4, 5
unit type, 15
universe, 41
upper semilattice, 19

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

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.