

# Reflections on Prismatic Constructions

Marc Coiffier

## Contents

The pure Calculus of Construction . . . . .	1
The Limits of Constructions . . . . .	2
Intuitionistic Booleans . . . . .	2
Sameness (aka. Identity) . . . . .	3
Putting it all together . . . . .	3
Inductive Types . . . . .	3
The $\mu$ Combinator : Lambda-constructions with extra steps . . . . .	4

The Calculus of Prismatic Constructions, upon which this platform is based, is an extension of the standard [CoC](#) with a mechanism for discriminating inductive constructors.

## The pure Calculus of Construction

It is already very well-described elsewhere, so I won't try to provide a full and correct history of the CoC. Suffice to say that it is a logically consistent programming language, that can prove properties within the framework of intuitionistic logic.

At its simplest, it provides five basic constructions :

- universes, of the form  $Set_n$ , are the “types of types”.  $Set_{n+1}$  is the type of  $Set_n$
- products, noted  $\forall(x : X), Y\ x$  – or  $X \rightarrow Y$  when  $Y$  doesn't depend on  $x$  – are the “types of functions”.  $\mathbb{N} \rightarrow \mathbb{R}$ , for instance is the type of functions from the natural numbers to the real numbers.
- functions or lambdas, noted  $\lambda(x : X), Y\ x$ , are the “proofs of products”. A valid lambda can be interpreted as the proof of a property, quantified over its variable.

- hypotheses, or variables, are the symbols introduced by surrounding quantifiers ( $\lambda$  and  $\forall$ ). In their context, they are valid proofs of their type.

For example, the identity function can be written  $\lambda(A : Set_0), \lambda(a : A), a$ , and it is a valid proof of  $\forall(A : Set_0), \forall(a : A), A$ , since  $a$  is a valid proof of  $A$  in its context.

- Applications, of the form  $f x$ , where  $f : \forall(x : X), Y x$  and  $x : X$ , signify the specialization of a quantified property over an object  $x$ .

For instance, given a proof  $f$  of  $\forall(x : \mathbb{N}), \exists(y : \mathbb{N}), y = x + 1$ , we can prove that  $\exists(y : \mathbb{N}), y = 10 + 1$ , by applying  $f$  to 10 (aka.  $f\ 10$ ).

Given these axioms, we can build many theorems and their proofs, in a verifiable manner (i.e. there exists an algorithm to automatically check whether a claim like  $x : X$  holds).

However, it's been known for a while that the CoC by itself is not capable of handling a large class of the proofs that modern mathematicians (and even ancient ones) take for granted.

## The Limits of Constructions

To illustrate the kind of reasoning that can't be carried out with raw intuitionistic logic, let's take an obvious statement : a boolean is either equal to true or to false.

We'd like to prove this statement using only the tools given by the CoC. For this, we have to define a few concepts, namely our Booleans, *true* and *false*, and what it means for two things to be equal.

### Intuitionistic Booleans

In order for two things to be considered the same, they must at least belong to the same family. In this case, it means that *true* and *false* must have the same type. By convention, we'll call the type of "true or false" the *Boolean* type, in honor of George Boole.

Given a Boolean  $b$ , we would like to be able to return different values from a function, depending on whether  $b$  is true or false. Otherwise, our Boolean wouldn't be much use in a computation.

With all that in mind, here is the definition I propose the *Boolean* type :

$$Boolean \equiv \forall(P : Prop)(ptrue : P)(pfalse : P), P$$

That is, a Boolean is a way to produce any  $P$ , given two alternatives *ptrue* and *pfalse*, and nothing else.

There are, intuitively, only two distinct ways to construct a closed Boolean term, given the above definition :

- $true \equiv \lambda(P : Prop)(ptrue : P)(pfalse : P).ptrue$
- $false \equiv \lambda(P : Prop)(ptrue : P)(pfalse : P).pfalse$

Our goal in the following sections will be to try and confirm this intuition, by proving it in the CoC.

### Sameness (aka. Identity)

Two values  $x$  and  $y$  can be said to be the same (when it comes to proving theorems) if everything that can be proven of  $x$  can also be proven of  $y$ . More formally, given a type  $A$  of things, and two values  $x$  and  $y$  of type  $A$  we have :

$$(x = y) \equiv \forall(P : A \rightarrow Set_n), Px \rightarrow Py$$

We can easily prove some intuitive properties for the  $=$  relation, such as :

- reflexivity :  $(x = x)$ , as proven by  $\lambda(P : A \rightarrow Set_n)(p : Px).p$
- symmetry :  $(x = y) \rightarrow (y = x)$ , proven by  $\lambda(e : x = y)(P : A \rightarrow Set_n)(py : Py), e(\lambda(a : A).Pa \rightarrow Px)(\lambda(px : Px).px) py$
- transitivity :  $(x = y) \rightarrow (y = z) \rightarrow (x = z)$ , as proven by  $\lambda(e_1 : x = y)(e_2 : y = z)(P : A \rightarrow Set_n)(px : Px).e_2 P(e_1 P px)$

### Putting it all together

We now have everything we need to prove that every Boolean is either *true* or *false*. First, let's formally state that property :

$$\forall(b : Boolean), (b = true) \cup (b = false) \equiv \forall(b : Boolean)(P : Set_n), (b = true \rightarrow P) \rightarrow (b = false \rightarrow P)$$

### Inductive Types

Inductive types can be described as enumerations of constructors. In Coq (and similarly in other proof assistants), an inductive type must be declared along with its constructors, using a syntax like :

```

Inductive T : forall A..., Type :=
| t0 : forall x0..., T (f0... x0...)
...
| tn : forall xn..., T (fn... xn...)
.

```

Here, we declare the inductive type  $T : \forall A..., Type$ , and its constructors called  $t_i$  ( $i \in \{0..n\}$ ).

As a more concrete example, here is how the type of Booleans can be defined inductively :

```

Inductive Boolean : Type := true : Boolean | false : Boolean.

```

The above definition is essentially a formal statement of the following description of Booleans : a Boolean can have one of two shapes, *true* or *false*, and cannot be any other thing.

This means that, if we want to prove a property  $Px$  for some unknown Boolean  $x$ , all we need is to prove  $Ptrue$  and  $Pfalse$ .

This exact information is summed up in what we call the *induction principle* for Booleans. In Coq, it will be given the name `Boolean_rect`, for instance, and have the type  $\forall(P : Boolean \rightarrow Type), Ptrue \rightarrow Pfalse \rightarrow \forall(b : Boolean), Pb$ .

## The $\mu$ Combinator : Lambda-constructions with extra steps

$$\mu(x) \equiv \mu_{\emptyset}(x)$$

$$\mu_{\Gamma}(\lambda(x : T).y) \equiv \mu_{\Gamma, (x:T)}(y)$$

$$\mu_{\Gamma}(Hx...) \equiv \lambda^* \Gamma^{\uparrow}. \Gamma^{\uparrow}[Hx]...$$

$$\emptyset^{\uparrow} \equiv \emptyset(\Gamma, (x : T_{\Gamma}))^{\uparrow} \equiv \Gamma^{\uparrow}, (x : T_{\Gamma^{\uparrow}})$$