# Reflections on Prismatic Constructions

Marc Coiffier

## Contents



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 withing 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. $f\,10$).

## 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 $P\,x$ for some unknown Boolean $x$, all we need is to prove $P\,true$ and $P\,false$.

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 $\forall(P : Boolean \rightarrow Type),\ P\,true \ \rightarrow\ P\,false \ \rightarrow\ \forall(b : Boolean),\ P\,b$.