# The Long Tale of Implementing CIC over CoC

Technical Article

Paul Lyutko, Groupoid Infinity

Kyiv 2016

## Contents

# 1 Introduction: The Why

Ladies and gentlemen! Let us start from the beginning. In the beginning there were quantifiers and lambdas binding variables in the terms. This is what any Pure Type System is built from. It can also be seen as a category made from types (as objects) and functions (as morphisms). If we want we can extend this category with any constructions we need but does it really need to be extended? What if it already contains all kinds of structures we know? Hereby we claim — yes, it contains. The structures we seek are inductive types given by their constructors and eliminators, and coinductive types, and higher inductive types, and also plenty of types formed by appropriate adjoint functors.

Such approach allows us to implement a full-featured language (EXE compiler) with dependent types atop of a small logical core of pure lambda calculus (OM compiler). You know the LISP language uses "code as data" metaphor, but we use "data as code" insteed!

It this text we discuss an encoding. An encoding is a model of one type-theory-like formal system in another type-theory-like formal system. The encoding we use implements axioms of Calculus of Inductive Construction (CIC), the type theory of inductive types, in Calculus of Construction, a Pure Type System (PTS). We do it by mapping types not to types but to structures known as setoids. Setoids provide with a well-behaved equality on our types and on the whole category. We also utilize the fact that this category has limits of functors going into it, a categorical notion refining dependent function type from PTS.

Attempts to explore encodings are not new. An historical example is the Church encoding for natural numbers and its generalizations [Berarducci]. Nowadays there are pragmatically charged compilers using them [Gonzalez]. We introduce an encoding capable of implementing dependent eliminator a.k.a. induction principle built by refining the Berarducci encoding with categorical limits. Attention: we do not extend the preexistent category of types, we just discover all required kinds of structures in the category of setoids. Here it goes. Enjoy!

# 2 Steps: The How

In this section we describe all the constructions used. Explanations are intended to be read by software engineers (really?). If you are a mathematician, just read the phrase "application of General Adjoint Functor Theorem to type theory" and meditate on it.

## 2.1 Forall and Lambda

If you are used to lambda expressions in functional programming you will find the following familiar.

The mathematical language we use is called Calculus of Constructions.

It is a typed programming language with 5 constructions: variable names, anonymous functions, type declarations, function calls/argument applications and predefined types.

Variables can be only defined as parameters to functions or parameters to type declarations. There are no stand-alone variables, every variable is a parameter somewhere. Moreover, functions and types always have one argument. There are no functions or types without arguments, and they cannot have more than one.

Also, there are no "normal" data: all the data is represented by functions.

From now on, we will use mathematical names for the constructions of the language.

Anonymous functions are called lambda-abstractions or simply lambdas, and are denoted by $\lambda$.

Type declarations are denoted by $\forall$ and are called function types (remember, everything is a function). Sometimes $\forall$ doesn't use the variable it declares, so we will use $\rightarrow$ as a shortcut.

Predefined types are called universes. In original Calculus of Constructions there were only two universes, an asterisk and a square. But our variation has many, and for each universe there is a "larger" universe, so they form an infinite sequence. We will use `*0`, `*1`, `*2` and so on. As very large universes are rarely needed, later we will introduce special names for few first universes:

```
def Prop := *0
def Type := *1
def Kind := *2
```

As both types and functions have formal parameters, there is a way to pass the actual parameters. Unlike most normal programming languages, when passing value arguments are denoted by foo(bar) and passing type arguments is denoted by foo<bar>, in CoC they are both written as foo bar, and foo bar baz means (foo bar) baz, i.e. foo bar "returns" something that can be "called" again.

For those mathematically inclined, Calculus of Construction is a Pure Type System, and Pure Type Systems are typed lambda-calculi.

Its language consists of variable references, functions (known as lambda abstractions, denoted by $\lambda$), function types (denoted by $\forall$) and applications of an argument to a function (denoted by juxtaposition). There are also universes (types of types). There are no others expressions! We will call the expressions of our language *terms*.

We are not going to introduce them formally by so-called "rules of inference" notation. Instead, we will show generic examples.

**Typing** Notation `a : A` means the term `a` is of type `A`.

**Abstraction** If there is a term `y : Y` containing variable `x : X`, then there is a new term `(λ x : X) → y` of type `(∀ x : X) → Y`. The former is the term of dependent function, the latter is its type. Also we write `A → B` as shorthand for `(∀ a : A) → B` if the variable `a` does not appear in term `B` — it is the well-known type of ordinary (non-dependent) functions.

**Application** If `f : (∀ x : X)→ Y` and `v : X` then `(f v) : Y[v/x]`. Also we assume `((λ x : X)→ y) v` to be equal to `y[v/x]`. Here `f[v/x]` denote the substitution of the term `v` in place of all occurencies of the variable `x` in the term `f`.

**Universes** If `a : A` then `A : ∗`. Here we denote "type of types" as `∗` symbol. These are also called universes. But what is the type of `∗` itself? Can we let it be just `∗`? No, it appears that `∗ : ∗` causes paradoxes, so we have to consider different universes. We can consider just one "superuniverse" *BOX* with `∗ : BOX` (then the whole system will be exactly CoC), or else we can provide an infinite hierarchy of similar universes of level $n$, denoted `∗{`$n$`}`, with the next universe containing the previous one `∗{`$n$`} : ∗{`$n+1$`}`, and it is the more general PTS commonly used.

**Predicativity** When we have different universes, we should ask what is the universe of the type of functions between given types. Let `M : ∗{`$m$`}` and `N : ∗{`$n$`}` and assume `(M → N) : ∗{`$r$`}`. There are two simple answers: $r = max(m, n)$ (the case is called the *predicative* hierarchy) and $r = n$ (the case is called the *impredicative* hierarchy).

## 2.2 Logics

In CoC/PTS the universes are first-class citizens, and they can be used freely in functions and in function types. Such usage allows not only dependent types, e.g. "type of lists of the given length", but also logical propositions. In the "types as propositions" approach we consider the empty type as the logical FALSE, and any inhabited type as the logical TRUE, and inhabitants of types are to be seen as evidences or proofs. Therefore we are able to state requirements against elements of a type (predicates) by forming appropriate dependent types and to check (verify) them making terms of that type. For further explanation of so-called Curry-Howard correspondence see [111]. We just note that there is a complex bussiness of describing the whole world in the language of dependent types mirroring older *predicate calculus* known from the XIX century.

**Important** Any well-typed term in PTS is simultaneously a proof of the theorem stated by its type.

## 2.3 Contexts: Curried Records

A context is a notion from type theory meaning a sequence of types of variables needed to formally define a constraint on binding of all free variable in a term. For example, $(T : *, x : T)$ declares a context including two variables where the type of the second variable is dependent on the value of the first one.

For the given context there is a notion of *tuple type* as a type of all sequences of terms of the given types. We need to use tuples (a.k.a. records) but PTSs have not tuples.

So this is the first layer of our encoding.

We simply shift our focus from types to context and functions between them. We can simply consider context as a tuple type. The pragmatic viewpoint allows such a shift because we are able to use types and terms in the high-level language being compiled to contexts and and sequences of terms in the low-level one. We denote tuple types as following:

```
record Pair : * :=
    (T : *)
    (x : T)
```

We can define a notion of a function between two contexts. For contexts `A` and `B`:

```
record A : * := (a₁ : A₁) (a₂ : A₂)
record B : * := (b₁ : B₁) (b₂ : B₂)
```

the type of such functions `A → B` is a context declaring function variables:

```
record FAB : * :=
    (f₁ : A₁ → A₂ → B₁)
    (f₂ : A₁ → A₂ → B₂)
```

All functions should receive all variables from the context $A$ and emit just one variable of the context $B$.

Also there is a well-known phenomenon in type theory — a curring. A curring means that function with a tuple argument is equivalent to a function with a number of arguments. Here we have curring by desing.

## 2.4 Setoids, Mappings

Recall the notion of equivalence. For any given type `T` an *equivalence relation* on `T` is a function of two arguments `eq : T → T → Prop` (written below in infix form as `a ∼ b`) satisfying the following three properties:

**Reflexivity** (For any `t₀ : T`) `t₀ ∼ t₀`

**Transitivity** (For any `t₁, t₂, t₃ : T`) If `t₁ ∼ t₂` and `t₂ ∼ t₃` then `t₁ ∼ t₃`.

**Symmetry** (For any `t₁, t₂ : T`) If `t₁ ∼ t₂` then `t₂ ∼ t₁`.

A tuple of the type and the equivalence relation on it is called a *setoid*. Formally:

```
record SetoidType : * :=
    (El : *)
    (Equ : El → El → Prop)
    (Refl : ∀{e₀ : El}, Equ e₀ e₀)
    (Trans : ∀{e₁ e₂ e₃ : El}, Equ e₁ e₂ → Equ e₂ e₃ → Equ e₁ e₃)
    (Sym : ∀{e₁ e₂ : El}, Equ e₁ e₂ → Equ e₂ e₁)
```

We call `El` the *carrier* of the setoid and `(Equ,Refl,Trans,Sym)` the *structure* on `El`. We are going to call the properties of equivalence the *axioms* of setoid. When the structure is known we call equivalent elements of the setoid carrier just *equal* and denote it `a = b`.

Also we need a notion of a function-like object between setoids. A *mapping* $m : (T_1, eq_1) \to (T_2, eq_2)$ is a tuple $(f, f_{OK})$ where $f : T_1 \to T_2$ is a function between carriers and $f_{OK} : \forall(t, t' : T_1) \to (t = t') \to (ft = ft')$ is just a predicate ensuring that any pair of equal elements of domain is mapped by $f$ onto a pair of equal elements of codomain. From the constructive viewpoint, the $f_{OK}$ maps evidences of equivalence $eq_1$ onto evidences of equivalence $eq_2$.

To speak simplified we can say that a mapping $m$ is just a function respecting the equality (the setoid structure).

Note that we can compile our types and functions of the high-level language to setoids and mapping of the low-level language.

**Kinds of equalities** We can see that setoid can couple a type with any equivalence. Given type $T$ there is a minimal equivalence on $T$ and it is called a propositional equality. However such equalities is dufficult to use in runtime, because we cannot pattern match against propositions. So for runtime testing of values to be equal we need a function of type $T \to T \to Bool$ which is called a decidable equality.

## 2.5 Categories, Functors

In the above we have considered the three different environments: types and functions between them, contexts and functions between them, setoids and mappings between them. All of them are usable as hosts for compiling lambda calculus based languages. Now we define such kind of environments more explicitly.

A *category* (as viewed by the type theory) is a tuple $(Ob, Hom, Id, Mul, UnitL, UnitR, Assoc)$ where $Ob : Type$ is the type of *objects*, $Hom : \forall(a, b : Ob) \to$ SetoidType is the family of setoids of *morphisms* denoted by $a \Rightarrow b$, $Id : \forall(o_0 : Ob) \to (o_0 \Rightarrow o_0)$ is a family of the *identity* morphisms $Ido_0$ defined for each object $o_0$, $Mul : \forall(o_1, o_2, o_3 : Ob) \to (o_2 \Rightarrow o_3) \to (o_1 \Rightarrow o_2) \to (o_1 \Rightarrow o_3)$ is a family of operation of *multiplication* (or *composition*) defined for any triples of objects, and predicates $UnitL$, $UnitR$ ensure that the identity is neutral against the multiplication, and $Assoc$ is the predicate ensures that the multiplication is associative.

In other words, a category comprises its carrier — all the objects and the morphisms between them, denoted by $m : a \Rightarrow b$. Thereafter we have morphisms between objects and equalities between morphisms. The carrier is accompanied by *the structure* of associative multiplication with neutral identities. The basic example is the category of setoids, denoted $Set$. Types and contexts equipped by minimal equivalence relations on them can be viewed as subcategories of $Set$.

A wise reader should ask now: what are functions between categories? The answer is a notion of *functor*. A functor $F : C \to D$ between given categories $C = (Ob_C, Hom_C, Id_C, Mul_C, UnitL_C, UnitR_C, Assoc_C)$ and $D = (Ob_D, Hom_D, Id_D, Mul_D, UnitL_D, UnitR_D, Assoc_D)$ is a tuple $(F_{OB}, F_{HOM}, F_{ID}, F_{MUL})$ where $F_{OB} : Ob_C \to Ob_D$ is a function between types of objects (we write simply $Fo$ for its action on objects), $F_{HOM} : \forall(a, b : Ob_C) \to (a \Rightarrow_C b) \to ((F_{OB}a) \Rightarrow_D (F_{OB}b))$ is a family

of mappings between setoids of morpisms (we write simply $Fm$), $F_{ID} : \forall(a : Ob_C) \rightarrow (Id_D(Fa) = F(Id_C a))$ is a predicate ensuring that the functor respects the identities, and $F_{MUL}$ is a predicate ensuring that the functor respects the multiplication (try to write its type by your own).

**Important** Consider the process of compilation from the high-level language with particular category of types to the low-level language with the other category. The procedure of encoding is just a change of the category we focus on when compiling our code. The technical information needed to compile is the definition of the new category from the old one.

# 6 References

## References


[1] Henk Barendregt *The Lambda Calculus. Its syntax and semantics* 1981

[2] Henk Barendregt *Lambda Calculus With Types* 2010

[3] Erik Meijer, Simon Peyton Jones *Henk: a typed intermediate language* 1984

[4] Per Martin-Löf *Intuitionistic Type Theory* 1984

[5] ????

[6] ????