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 engeneers (really?). If you are a mathematician, just read the phrase "application of General Adjoint Functor Theorem at type theory" and meditate on it.

2.1 Forall and Lambda

If you used to use lambda expressions in functional programming you will find the following familiar. All our work is performed in a particular typed lambda calculus, specifically in PTS, a Pure Type System. The simplest of them is CoC, the Calculus of Constructions. Its language consists of variable references, function expressions (known as lambda abstractions, denoted by λ), their types (known as types of functions or dependent functions, denoted by $\forall$) and applications of an argument to a function (denoted by juxtaposition). There are also universes (types of types). No others expressions! Expression of our language we will call *terms*. Here we will not introduce it fully formal by so-called "rules of inference", but rather show generic examples.

Typing Notation $a : A$ means term a is of type A.

Abstraction If there is a term $y : Y$ containing variable $x : X$, then there is a new term $(\lambda x : X) \to y : (\forall x : X) \to Y$. The former is the term of dependent function, the latter is its type. Also we write $A \to B$ for shorthand of $(\forall a : A) \to B$ if the variable a does not appear in term B (it is the well-known type of non-dependent functions).

Application If $f : (\forall x : X) \to Y$ and $v : X$ then $(fv) : Y[v/x]$. Also we assume $((\lambda x : X) \to y)v$ to be equal to $y[v/x]$. Here $f[v/x]$ means 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 (than 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, and this is more general PTS.

Predicativity If 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 $(M \to N) : *r$. There are two simple answers: $r = max(m, n)$ (called the predicative hierarchy) and $r = n$ (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.

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-Lf *Intuitionistic Type Theory* 1984

[5] ????

[6] ????