# Introduction to the Calculus of Inductive Definitions

Christine Paulin-Mohring[1]

LRI, Univ Paris-Sud, CNRS and INRIA Saclay - Île-de-France, Toccata, Orsay F-91405
Christine.Paulin@lri.fr

## 1 Introduction

The Calculus of Inductive Constructions (CIC) is the formalism behind the interactive proof assistant Coq [18, 3]. It is a powerful language which aims at representing both functional programs in the style of the ML language family and proofs in higher-order logic. Many data-structures can be represented in this language: usual data-types like lists and binary trees (possibly polymorphic) but also infinitely branching trees. At the logical level, inductive definitions give a natural representation of notions like reachability, operational semantics defined using inference rules...

Inductive definitions as a basis of a proof language were formalized in the early 90's in two different contexts. The first one is Martin-Löf's Type Theory [13]: this theory was originally presented with a set of rules defining basic notions like products, sums, natural numbers, equality... All of them (except for functions) are an instance of a general scheme of inductive definitions which was studied by P. Dybjer [9]. The second one is an extension of the pure Calculus of Constructions. In the Calculus of Constructions, there is a general impredicative product primitive which is powerful enough to encode inductive definitions [4, 17], however this encoding has some drawbacks: efficiency of computation of functions over these data-types and some properties that cannot be proven. The extension of the formalism with primitive inductive definitions [8, 15] was consequently a natural choice. In proof assistants based on HOL (higher-order logic), an impredicative encoding of inductive definitions is used: this is made possible by the existence of a primitive infinite type (including integers) and the fact that HOL is only concerned by extensional properties of objects (not computations) [16].

## 2 Proof System

Inductive definitions are added on top of a pure type system (PTS) which is given by a set of sorts (Prop and $\mathsf{Type}_i$) and term constructors for dependent type product $(\forall x, B)$, abstraction $(\mathbf{fun}\, x \Rightarrow t)$ and application $(t\, x_1 \ldots x_n)$. There is no syntactic distinction between types and terms. Types are used to represent both sets of objects and logical properties. termes are used for represnting both objects and proofs depending on their type.

A new inductive definition can be added to the environment: it requires to specify its name, its arity (the type of the inductive definition) and the set of its *constructors*. For instance the following definitions introduce successively booleans, unary natural numbers, the transitive closure of a relation $R$ and the polymorphic equality.

```
Inductive bool : Type := true | false.
Inductive nat : Type := O | S : nat → nat.
Inductive RT A (R : A → A → Prop) : A → A → Prop :=
  RTrefl:∀ x, RT A R x x.
| RTR:∀ x y, R x y → RT A R x y.
| RTtran:∀ x y z, RT A R x z → RT A R z y → RT A R x y.
Inductive eq A (x:A) : A → Prop := eqrefl : eq x x.
```

The general pattern for a (mutually) inductive definition is

```
Inductive I_1 pars : Ar_1 := ...
  | c : ∀ (x_1:A_1)..(x_n:A_n), I_1 pars u_1..u_p
  ...
with I_2 pars : Ar_2 := ...
with ...
```

We introduce some terminology

- $pars$ are called the *parameters* of the inductive definition and will be the same for all definitions;
- $Ar_j$ is called the *arity*;
- $u_i$ is an *index*;
- $\forall(x_1 : A_1)..(x_n : A_n), I_1\, pars\, u_1..u_p$ is a *type of constructor*
- $A_i$ is a *type of argument* of constructor

There are conditions to accept that the definition is well-formed:

- Arities are of the form $\forall(y_1 : B_1)..(y_p : B_p), s$, with $s$ a sort which is called the *sort* of the inductive definition.
- Type of constructors $C$ are well-typed:

$$(I_1 : \forall pars, Ar_1)..(I_k : \forall pars, Ar_k)\,(pars) \vdash C : s$$

  - if $s$ is predicative (not Prop) then type of arguments of constructors are in the same universe: for all $i$, $A_i : s$ or $A_i$ : Prop
  - if $s$ is Prop, we distinguish between *predicative* definitions where $A_i$ : Prop for all $i$ and *impredicative* definitions where there is at least one $i$ such that $A_i$ : Type.

There is also a positivity condition: occurrences of $I_j$ should only occur strictly positively in types of arguments of constructors $A_i$ which means that we are in one of these cases:

- non-recursive case: $I_j$ does not occur in $A_i$
- simple case: $A_i = I_j\, t_1 \ldots t_p$ (not necessarily the same parameters, $I_j \notin t_k$)
- functional case: $A_i = \forall z : B_1, B_2$ with $I_l \notin B_1$ and $I_j$ strictly positive in $B_2$
- nested case: $A_i = J\, t_1 \ldots t_p$ with $J$ another inductive definition with parameters $X_1 \ldots X_r$. When $t_1 \ldots t_r$ are substituted for $X_1 \ldots X_r$ in the types of constructors of $J$, the strict positivity condition should still be satisfied.

The language of the PTS is extended with access to the inductive definition and its constructors plus two new constructions for pattern-matching and fixpoint.

The inductive definition itself is a new constant, its type is given by its arity and is generalized with respect to the parameters.

$$I_l : \forall pars, Ar_l$$

The Calculus of Constructions follows the logical rules of natural deduction where each concept is associated with introduction and elimination rules. A computation rules explains how a combination of introduction and elimination rules for the same notion (a cut) can be eliminated.

In the case of inductive definitions, introduction rules are given by the constructors.

Given that $c$ is the $i$-th constructor of an inductive definition $I$ with parameters $pars$ and type of constructor $C$, we have:

$$c \equiv \texttt{Constr}(i, I) : \forall pars, C$$

Elimination rules uses two different notions: a pattern-matching rule extended for dependent types (each branch can have a different type, depending on the constructor) and a (restricted) fixpoint construction for recursive definitions.

The primitive rule for pattern-matching comes in a very primitive way: it covers one level of constructors and should be complete (one branch for each constructor):

$$\frac{\begin{array}{l} t : I\,pars\,t_1 \ldots t_p \\ y_1 \ldots y_k, x : I\,pars\,y_1 \ldots y_k \vdash P(y_1 \ldots y_k\,x) : s' \\ (x_1 : A_1 \ldots x_n : A_n \vdash f : P(u_1 \ldots u_k\,(c\,x_1 \ldots x_n)))_c \end{array}}{\begin{array}{l} \mathbf{match}\,t\,\mathbf{as}\,x\,\mathbf{in}\,I\,\_\,y_1 \ldots y_k\,\mathbf{return}\,P(y_1 \ldots y_k, x) \\ \mathbf{with} \ldots \mid c\,x_1 \ldots x_n \Rightarrow f \mid \ldots \\ \mathbf{end} : P(t_1 \ldots t_p\,t) \end{array}}$$

The reduction rule (called $\iota$) applies when $t$ starts with a constructor and is as expected (reduces to the corresponding branch after instantiating the pattern variables with the arguments of the constructor).

The main restriction lies in the relation between the sort $s$ of the inductive definition and the sort $s'$ of the pattern-matching.

When $s$ is Type, which means that we have a predicative inductive definition, then we can have any possible sorts $s'$ for case analysis.

When $s$ is Prop however, the question is a bit more tricky for several reasons:

- Prop is an impredicative sort, so uncareful elimination can easily introduce paradoxes;
- it is sometimes useful to add an axiom of proof irrelevance for propositions (which says that two different proofs of the same property can be considered as equal) so while it is good to be able to prove that for instance `true` $\neq$ `false`, a similar mechanism that will lead to two terms (representing proofs of) in $A \vee B$ that are provably different is less desirable;
- Prop is used for program extraction: any term in $A$ : Prop is removed during extraction so should not be needed for computing the informative art, in a pattern-matching is done on a term in an inductive definition in Prop, but with the result being used for computing, then we need to be able to execute the match without executing the head, which is only feasible in specific cases.

For an inductive definition of sort Prop, the only elimination allowed is on the sort Prop itself. There are exceptions where any elimination is allowed: in the specific case where $I$ is a *predicative* definition with only zero or one constructor with all its arguments $A_i$ : Prop. The exception covers cases like absurdity, equality, conjunction of two propositions, accessibility...

Fixpoint constructions in Coq are mainly introduced via global declarations.

`Fixpoint f (`$x_1$`:`$A_1$`)... (`$x_m$`:`$A_m$`){struct `$x_n$`}:B:=t.`

they correspond to an internal fixpoint construction

`fix f (`$x_1$`:`$A_1$`)... (`$x_n$`:`$A_n$`):∀(`$x_{n+1}$`:`$A_{n+1}$`)... (`$x_m$`:`$A_m$`)B := fun `$x_{n+1}$`... `$x_m$` ⇒ t.`

In general, an expression **fix** $f(x_1 : A_1) \ldots (x_n : A_n) : B := t$ is well typed of type $B$ when

- $t$ is well-typed of type $B$ in an environment containing $(f : \forall(x_1 : A_1) \ldots (x_n : A_n), B)$ and $(x_1 : A_1) \ldots (x_n : A_n)$;
- $t$ satisfies an extra syntactic condition that recursive calls to $(f\,u_1 \ldots u_n)$ in $t$ are made on terms $u_n$ *structurally* smaller than $x_n$.

The reduction rule is the usual fixpoint reduction except that in order to avoid infinite loops, it is only activated when the $n$-th argument of the fixpoint starts with a constructor.

## 3 Proof-Theoretical Properties

Proof theoretical properties of systems like the Calculus of Constructions are complex to perform in full detail, first because these systems are logically powerful (due to the impredicativity, the hierarchy of universes, the type dependency) and second because there are many syntactic properties to be established like the Church-Rosser property or subject-reduction which are made even more complicated because of the general pattern of inductive definitions. Several proofs covering subsystems of Coq exists, Bruno Barras in his thesis formalized and proved meta-theoretical properties (including typing decidability assuming normalization) of a Calculus of Constructions with Inductive Definitions.

## 4 Pragmatic Properties

Inductive definitions are an appropriate formalism to introduce complex data-structures without extra encoding and to define recursive functions on these data. Because of type dependency, it is even possible to embed specification parts inside the type, leading to a very precise description. Computation is part of the Coq kernel, many efforts have been made to make it more efficient using compiler technologies. With inductive definitions, the Coq language contains a mini ML sublanguage (with no effect and only terminating functions). It is convenient for formalizing complex programs which can then be proven, the most impressive example being the CompCert project of a optimizing compiler for C programs [12]. A second benefit is the ability to implement inside Coq various decision procedures and then to prove a scheme of reflection which allows to use the result of the execution of the procedure in order to build proofs of complex facts. The (partial)-correctness of the procedure is proven once and then the proof of any instance of the problem is reduced to a simple computation. This principle can be used for complex procedures but also for simple reasoning steps. The popular Ssreflect [11] (for small scale reflection) environment (including a tactic language and libraries) which has been successfully used for formalizing in Coq the four color theorem and the Feit-Thompson theorem use intensively this computational capability of the Coq system mainly on the type of booleans.

Inductive definitions are also the basis of a more declarative style of specifications. Inductive definitions of families are a natural way to encode relations like reachability, or semantics of programming languages or transition systems. Proofs can be done using a resolution-like mechanism.

## 5 Proof Applications

Coq is developed for more than 30 years now and there has been a lot of impressive examples formalized using it.

Many interesting proofs combine advanced algorithms and non-trivial mathematics like the proof of the four-color theorem by Gonthier & Werner at INRIA and Microsoft-Research [10], a primality checker using Pocklington and Elliptic Curve Certificates developed by Théry et al. at INRIA [19] and the proof of a Wave Equation Resolution Scheme by Boldo et al. [5]. Coq can also be used to certify the output of external theorem provers like in the work on termination tools by Contejean and others [7], or the certification of traces issued from SAT & SMT solvers done by Grégoire and others [1]. Coq is also a good framework for formalizing programming environments: the Gemalto and Trusted Logic companies obtained the highest level of certification (common criteria EAL 7) for their formalization of the security properties of the JavaCard platform [6]; as mentioned earlier Leroy and others developed in Coq a certified optimizing compiler for C (Leroy et al.) [12]. Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [2]. G. Morrisett and others also developed on top of Coq the Ynot library for proving imperative programs using separation logic [14]. These represent typical examples of what can be achieved using Coq.

## 6 Trends and Open Problems

The current inductive definitions of Coq present certain drawbacks. The syntactic condition for accepting fixpoints is very sensitive and not well-suited when developing a proof using tactics. Different approaches using type annotations have been proposed instead but none of them is yet available for Coq. Also the primitive pattern-matching is not the natural expected rule when dealing with elimination of a particular instance of an inductive definition, where you expect some cases to disappear or to be partially instantiated.

In systems like Coq there always is a trade-off between keeping the language and the kernel small enough to ensure correctness and use encodings for more high-level constructions or include these constructions directly in the language.

In general the defined equality in the Calculus of Inductive Constructions does not have all the expected properties. The current work on Homotopy Type Theory [20] is an attempt to solve this problem. It includes a notion of generalized inductive definitions where the equality definition is included in the declaration, making the definition of quotient types more direct.

## 7 Conclusions

The Calculus of Inductive Constructions provides a powerful language for the interactive development of proofs and programs. It includes a mini functional programming languages that is sufficient for programming complex data-structures and programs. The specification language itself can use a declarative style with (almost) no limit to the expressiveness.

## References


1. M. Armand, G. Faure, B. Grégoire, Ch. Keller, L. Théry, and B. Werner. A Modular Integration of SAT/SMT Solvers to Coq through Proof Witnesses. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs, CPP 2011*, Lecture Notes in Computer Science. Springer, 2011.
2. G. Barthe, B. Grégoire, and S. Zanella Béguelin. Formal certification of code-based cryptographic proofs. In *36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009*, pages 90–101. ACM, 2009. See also: CertiCrypt `http://www.msr-inria.inria.fr/projects/sec/certicrypt`.
3. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
4. C. Böhm and A. Berarducci. Automatic synthesis of typed $\lambda$-programs on term algebras. *Theoretical Computer Science*, 39, 1985.
5. S. Boldo, F. Clément, J.-C. Filliâtre, M. Mayero, G. Melquiond, and P. Weis. Formal Proof of a Wave Equation Resolution Scheme: the Method Error. In Matt Kaufmann and Lawrence C. Paulson, editors, *Proceedings of the first Interactive Theorem Proving Conference*, volume 6172 of *LNCS*, pages 147–162, Edinburgh, Scotland, July 2010. Springer. Extended version `http://hal.inria.fr/hal-00649240/PDF/RR-7826.pdf`.
6. Boutheina Chetali and Quang-Huy Nguyen. About the world-first smart card certificate with EAL7 formal assurances. Slides 9th ICCC, Jeju, Korea, September 2008. `www.commoncriteriaportal.org/iccc/9iccc/pdf/B2404.pdf`.
7. E. Contejean, A. Paskevich, X. Urbain, P. Courtieu, O. Pons, and J. Forest. A3pat, an approach for certified automated termination proofs. In John P. Gallagher and Janis Voigtländer, editors, *Proceedings of the 2010 ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation, PEPM 2010*, pages 63–72. ACM, 2010.
8. Th. Coquand and C. Paulin-Mohring. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *Proceedings of Colog'88*, volume 417. Springer-Verlag, 1990.
9. P. Dybjer. Inductive families. *Formal Asp. Comput.*, 6(4):440–465, 1994.
10. G. Gonthier. Formal proof the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008. `http://www.ams.org/notices/200811/tx081101382p.pdf`.
11. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.

12. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
13. P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.
14. G. Morrisett and al. The Ynot project. `http://ynot.cs.harvard.edu/`.
15. C. Paulin-Mohring. Inductive Definitions in the System Coq - Rules and Properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi a nd Applications*, number 664, 1993. LIP research report 92-49.
16. L.C. Paulson. A fixedpoint approach to implementing (co)inductive definitions. In Alan Bundy, editor, *Proceedings of the 12th International Conference on Automated Deduction*, volume 814 of *LNAI*, pages 148–161. Springer-Verlag, 1994.
17. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442. Springer-Verlag, 1990. technical report CMU-CS-89-209.
18. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012. http://coq.inria.fr.
19. L. Théry and G. Hanrot. Primality proving with elliptic curves. In Klaus Schneider and Jens Brandt, editors, *TPHOLs*, volume 4732 of *Lecture Notes in Computer Science*, pages 319–333. Springer, 2007. See also: Certifying Prime Number with the Coq prover `http://coqprime.gforge.inria.fr/`.
20. The Univalent Foundations Program. *Homotopy Type Theory Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. `http://homotopytypetheory.org/book/`.