Introduction to the Calculus of Inductive Constructions

Christine Paulin-Mohring

Introduction to the Calculus of Inductive Constructions

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 [24, 5]. It is a powerful language which aims at representing both functional programs in the style of the ML language 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 and operational semantics defined using inference rules.

Inductive definitions in the context of a proof language were formalised in the early 90's in two different settings. The first one is Martin-Löf's Type Theory [18]. 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 has been studied by P. Dybjer [14]. The second one is the pure Calculus of Constructions. This is a typed polymorphic functional language, which is powerful enough to encode inductive definitions [6, 23], but this encoding has some drawbacks: efficiency of computation of functions over these data-types and some natural properties that cannot be proven. The extension of the formalism with primitive inductive definitions [13, 20] was consequently a natural choice. In proof assistants based on higher-order logic (HOL), 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) [22].

Inductive definitions, as a primitive or derived notion are one of the main ingredients of the languages for interactive theorem proving both for representing objects and logical notions.

In this paper, we give a quick overview of the Calculus of Inductive Constructions, the formalism behind the Coq proof assistant. In section 2, we present the language and the typing rules. We start with the pure functional part and then continue with the inductive declarations. We shall then briefly discuss the properties of this language in section 3, both from the theoretical and pragmatic points of view. We shall then conclude with examples of applications, the description of some of the trends in sections 4,5 and 6.

2 Proof System

2.1 The Calculus of Constructions

The Calculus of Constructions which is the purely functional language underlying the Calculus of Inductive Constructions has been introduced by Coquand and Huet [11, 12]. It can be defined as a pure type system (PTS). A PTS is a typed lambda-calculus with a unique syntactic language describing both terms and types. Terms include variables, (typed) abstractions (written $\mathbf{fun}\, x : A \Rightarrow t$) and applications (written $t\, u$) as in ordinary lambda-calculus. The type of an abstraction is a (dependent) product $\Pi x : A, B$ making possible for the type B of t to depend on the variable x. The notation $A \to B$ for the type of functions from type A to type B is just an abbreviation in the special case where B does not depend on x. Types are themselves typed objects, the type of a type will be a special constant called a *sort*. There is at least one sort called Type. Different PTS depend on the set of sorts we start with (each sort corresponds to a certain universe of objects) and also which products can be done (in which universes).

For instance, with A a type, the identity function $\mathbf{fun}\, x : A \Rightarrow x$ is a term of type $A \to A$. With A being a type variable, we may build the polymorphic identity $\mathbf{fun}\, A : \mathsf{Type} \Rightarrow \mathbf{fun}\, x : A \Rightarrow x$ of type $\Pi A : \mathsf{Type}, A \to A$.

In the case of the Calculus of Constructions, we have an infinite set of sorts $\mathcal{S} \stackrel{\text{def}}{=} \{\mathsf{Prop}\} \cup \bigcup_{i \in \mathbb{N}} \{\mathsf{Type}_i\}$. The sort Prop captures the type of expressions which denote logical propositions. We follow the Curry-Howard correspondence where a proposition A is represented by a type (namely the type of proofs of A) and a proof of the logical proposition A will correspond to an object t of type A. If A and B are two types corresponding to logical propositions, then the proposition $A \Rightarrow B$ will be represented by the type $A \to B$ of functions transforming proofs of A into proofs of B, the proposition $A \wedge B$ will be represented by the type $A \times B$ of pairs build with a proof of A and a proof of B. Given a type T, the type $\Pi x : T, B$ will represent the type of dependent functions which given a term $t : T$ computes a term of type $B[t/x]$ corresponding to proofs of the logical proposition $\forall x : T, B$. Because types represent logical propositions, the language will contain empty types corresponding to unprovable propositions.

Notations. We shall freely use the notation $\forall x : A, B$ instead of $\Pi x : A, B$ when B represents a proposition. We write $t[u/x]$ for the term t in which the variable x has been replaced by the term u. The term $t\, u_1 \ldots u_n$ represents $(\ldots (t\, u_1) \ldots u_n)$ and $\mathbf{fun}\, (x_1 : A_1) \ldots (x_n : A_n) \Rightarrow t$ (resp. $\Pi (x_1 : A_1) \ldots (x_n : A_n), B$) is the same as $\mathbf{fun}\, x_1 : A_1 \Rightarrow \ldots \mathbf{fun}\, x_n : A_n \Rightarrow t$ (resp. $\Pi x_1 : A_1, \ldots \Pi x_n : A_n, B$). The term $A \to B \to C$ should be understood as $A \to (B \to C)$ and $\Pi x : A, B \to C$ is the same as $\Pi x : A, (B \to C)$. We sometimes omit the type of the variable in abstractions and products when they are clear from the context.

In higher-order logic, propositions and objects are written using the same functional language with abstractions and applications. We may want to introduce a binary relation on a type A as a variable R. This variable will have type $A \to A \to \mathsf{Prop}$ (this type replaces an arity declaration in first-order logic). We can build a predicate $\mathbf{fun}\, x : A \Rightarrow R\, x\, x$ representing the set of objects which are in relation with themselves, this predicate will have type $A \to \mathsf{Prop}$ (this expression shall not be confused with the type $\forall x : A, R\, x\, x$ of type Prop, expressing the reflexivity of R).

We need $A \to A \to \mathsf{Prop}$ to be a well-formed type, which means it is typed with a sort. This sort will be Type_1. If we want to iterate constructions on Type_1, we shall need this sort itself to be well-typed, that will require introducing a new sort Type_2 to be the type of the object Type_1. The need for an infinite hierarchy of universes comes from the fact that the more naive system where we have only one sort Type of type Type is inconsistent.

The Calculus of Inductive Constructions manipulates judgements which are of the form

$$x_1 : A_1, \ldots, x_n : A_n \vdash t : A$$

In this judgement, $x_1 : A_1, \ldots, x_n : A_n$ is called the context and the part $t : A$ on the right-hand side of the $\vdash$ sign is called the conclusion. In the context, x_i is a variable declared of type A_i representing the name of an object. Following the propositions as types paradigm, when A_i denotes a logical proposition, x_i will be a name given to the hypothesis that A_i holds. The judgement can be read as: under the assumption that we have objects x_i of type A_i, the term t is well-formed of type A.

For instance, in order to reason on Peano integers, it is possible to introduce a signature with a type variable $N : \mathsf{Type}$ for representing the type of Peano integers, an object $z : N$ (for zero) and an object $S : N \to N$ for the successor function. We call Γ_S the context $N : \mathsf{Type}, z : N, S : N \to N$. The following judgements are derivable :

$$\Gamma_S \vdash z : N \qquad \Gamma_S \vdash S\, z : N \qquad \Gamma_S \vdash S\, (S\, z) : N$$

It is also possible to introduce a binary relations le which represents the natural order on N. We shall add $le : N \to N \to \mathsf{Prop}$ and hypotheses like $lez : \forall x : N, le\, z\, n$ and $leS : \forall x\, y : N, le\, x\, y \to le\, (S\, x)\, (S\, y)$. We introduce a new context

$$\Gamma_N \stackrel{\text{def}}{=} \Gamma_S, le : N \to N \to \mathsf{Prop}, lez : (\forall x : N, le\, z\, x), leS : (\forall x\, y : N, le\, x\, y \to le\, (S\, x)\, (S\, y))$$

We shall be able to derive

$$\Gamma_N \vdash lezz : le\,z\,z \qquad \Gamma_N \vdash leSz\,z\,(lezz) : le\,(S\,z)\,(S\,z)$$

It is easy, for every natural number n to find a term l_n such that $\Gamma_N \vdash l_n : le\,(S^n z)\,(S^n z)$. We can do it (at the meta level) via a simple induction on n. It would be nice to prove internally the property $\forall x : N, le\,x\,x$ but our context is too weak for that because it contains no induction property, so N might contain more objects than the ones written $S^n\,z$. In first-order logic, the induction principle for a property P is added for every possible property P, leading to an infinite set of axioms. In a higher-order logic like the Calculus of Constructions it is sufficient to add a single proposition as an axiom that will contain a quantification over all possible properties P. We introduce $\Gamma_P \stackrel{\text{def}}{=} \Gamma_N, ind : (\forall P : N \to \mathsf{Prop}, P\,z \to (\forall x : N, P\,x \to P\,(S\,x)) \to \forall x : N, P\,x)$ and we can derive the judgement

$$\Gamma_P \vdash ind\,(\mathbf{fun}\,x : N \Rightarrow le\,x\,x)(lez\,z)(\mathbf{fun}\,(x : N)(I : le\,x\,x) \Rightarrow leS\,x\,x\,I) : \forall x : N, le\,x\,x$$

Type-checking proof-terms can be tricky and cumbersome. The user usually does not want and does not need to do it, because proofs are built just interacting with the properties that have to be proven using high-level programs called tactics (see section 3.2). However, the proof-term is always build underneath by the system and given for type checking (which is also proof-checking) to a trusted kernel of the system that will guaranty the absence of flaw in the proof. This proof term can be checked independently by different programs and the information contained in the proof term can also be used for instance for analysing dependency and intelligent printing.

Formally a CIC judgement will be of the form $\Gamma \vdash t : A$ with Γ a context and t and A two terms (A being also a type). The weaker judgement $\Gamma \vdash$ states that the context Γ is well-formed.

The inference rules corresponding to the pure functional part of the Calculus of Constructions are given in figure 1, with $\mathcal{S} \stackrel{\text{def}}{=} \{\mathsf{Prop}\} \cup \bigcup_{i\in\mathbb{N}}\{\mathsf{Type}_i\}$.

$$\frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Prop} : \mathsf{Type}_1} \qquad \frac{\Gamma \vdash}{\Gamma \vdash \mathsf{Type}_i : \mathsf{Type}_{i+1}}$$

$$\frac{\Gamma \vdash \qquad x : A \in \Gamma}{\Gamma \vdash x : A} \qquad \frac{\Gamma \vdash A : s \quad x \notin \Gamma \quad s \in \mathcal{S}}{\Gamma, x : A \vdash}$$

$$\frac{\Gamma, x : A \vdash t : B}{\Gamma \vdash \mathbf{fun}\,x : A \Rightarrow t : \Pi x : A, B} \qquad \frac{\Gamma \vdash f : \Pi x : A, B \quad \Gamma \vdash a : A}{\Gamma \vdash f\,a : B[a/x]}$$

$$\frac{\Gamma, x : A \vdash B : \mathsf{Prop}}{\Gamma \vdash \Pi x : A, B : \mathsf{Prop}} \qquad \frac{\Gamma, x : A \vdash B : \mathsf{Type}_i \quad \Gamma \vdash A : \mathsf{Type}_i}{\Gamma \vdash \Pi x : A, B : \mathsf{Type}_i}$$

$$\frac{\Gamma \vdash t : A \quad \Gamma \vdash B : s \quad A \preceq B}{\Gamma \vdash t : B}$$

Fig. 1. Inference rules for the purely functional part of CIC

We comment some rules in figure 1. There are two different rules for typing a product $\Pi x : A, B$. In both cases the term B should be well-typed in a context where we have $x : A$ and its type should be a sort s. When s is Prop, then the product stays in Prop even if A itself lies in a bigger universe. So $\Pi X : \mathsf{Prop}, X \to X$ has type Prop. We say that Prop is an impredicative sort, because one can build new objects in Prop using a universal quantification on the class of all objects in Prop (including the one currently defined). Impredicative systems are powerful but also very fragile in the sense that impredicativity does not interact very well with other features leading rapidly to inconsistent systems. For instance we have Prop of type Type_1 but Prop only can be impredicative, the impredicativity of Type_1 gives an inconsistent system [10], so if we want to build the type $\Pi X : \mathsf{Type}_1, X \to X$ we need a bigger universe Type_2.

Another interesting rule is the last one, called the conversion rule. It says that a term of type A can be also seen as a term of type B, given B is well-formed and the relation $A \preceq B$ holds. This relation serves two purposes. First it implements the universe cumulativity, $\mathsf{Prop} \preceq \mathsf{Type}_1$ and $\mathsf{Type}_i \preceq \mathsf{Type}_{i+1}$: any term typed in one universe can be considered as an element of a bigger universe. Second, it implements the fact that types are considered modulo computation (namely β-equivalence in the current system). The β-reduction rule implements function computation namely the fact that an abstraction of a term t over a variable x applied to a term a behaves like the term t in which x has been replaced by a: $(\mathbf{fun}\, x : A \Rightarrow t)\, a \simeq t[a/x]$. An important aspect of this rule is that this is the same term which is typed by A and by B, so these computation steps are done automatically and do not leave traces in the proof-term. We shall come back to this feature in section 3.2.

Using this purely functional part, it is possible to encode many interesting notions. For instance $\forall C : \mathsf{Prop}, C$ is a logical proposition (a term of type Prop) which encodes absurdity ($\bot$) : there is no closed term of type $\forall C : \mathsf{Prop}, C$ (so no proof of $\bot$ without hypothesis) and also from a proof t of $\forall C : \mathsf{Prop}, C$ one can build a proof $t\, C$ of an arbitrary proposition C so the natural deduction rule for eliminating $\bot$ is derivable in the logic.

It is also possible to encode an existential quantification (using a universal quantification in a negative position)

$$\exists x : A, B \stackrel{\text{def}}{=} \forall C : \mathsf{Prop}, (\forall x : A, B \to C) \to C$$

Both the introduction and elimination rules for existential quantification in natural deduction (first column) are derivable (CIC typed terms in second column)

$$\frac{\Gamma \vdash B[t/x]}{\Gamma \vdash \exists x : A, B} \qquad \frac{\Gamma \vdash p : B[t/x]}{\Gamma \vdash \mathbf{fun}\, C (H : \forall x : A, B \to C) \Rightarrow H\, t\, p : (\exists x : A, B)}$$

$$\frac{\Gamma \vdash \exists x : A, B \quad \Gamma, B \vdash C \quad x \notin \Gamma, C}{\Gamma \vdash C} \qquad \frac{\Gamma \vdash t : \exists x : A, B \quad \Gamma, x : A, p : B \vdash u : C \quad x \notin \Gamma, C}{\Gamma \vdash t\, C\, (\mathbf{fun}\, (x : A)(p : B) \Rightarrow u) : C}$$

One has to be careful that CIC implements a constructive logic and not a classical one. So the stronger form of elimination of absurdity namely

$$\frac{\Gamma, \neg C \vdash \bot}{\Gamma \vdash C}$$

is not provable in general and also one can prove $\exists x, B \vdash \neg \forall x, \neg B$ but not the opposite direction. The meaning of an existential quantification in CIC is stronger than the one in classical logic in the sense that from a proof of $\exists x : A, B$ one will always be able to extract a term t such that $B[t/x]$ is provable while in classical logic one only get (via Herbrand's theorem) the existence of a finite number of terms $t_1, \ldots, t_k$ such that $B[t_1/x] \vee \ldots \vee B[t_k/x]$ is provable for existential formulas.

Higher-order quantification can also be used to represent logical relations. For instance Leibniz equality $x = y$ with x and y of type A can be encoded using the proposition $\forall P : A \to \mathsf{Prop}, P\, x \to P\, y$. The two rules for introduction and elimination are derivable in CIC

$$\frac{}{\Gamma \vdash t = t} \qquad \frac{\Gamma \vdash t = u \quad \Gamma, x : A \vdash B : \mathsf{Prop} \quad \Gamma \vdash B[t/x]}{\Gamma \vdash B[u/x]}$$

2.2 Inductive Definitions

Inductive definitions are introduced on top of the pure Calculus of Constructions. Their main purpose is to provide an efficient representation of data-types.

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*.

Examples For instance, natural numbers defined with two constructors z and S, can be introduced as before with the following declaration:

```
Inductive N : Type := z : N | S : N → N.
```

This declaration adds new typed objects $N : \mathsf{Type}, z : N, S : N \to N$ in the context. But unlike in the first-order case, the logic captures the fact that N is the *initial algebra* relatively to these two operations, consequently we shall be able to build functions of type $N \to A$ using the initially property and to derive properties like $\forall x : N, S\,x \neq z$ and $\forall x\,y : N, S\,x = S\,y \to x = y$. Also the induction principle is provable.

Inductive definitions are also used for defining relations. The order on natural numbers can be defined using the properties we gave as axioms before:

```
Inductive le : N → N → Prop :=
  lez:∀x, le z x
| leS:∀x y, le x y → le (S x) (S y).
```

This declaration introduces $le : N \to N \to \mathsf{Prop}, lez : (\forall x : N, le\,z\,x), leS : (\forall x\,y : N, le\,x\,y \to le\,(S\,x)\,(S\,y))$ in the context, we shall also be able to prove the fact that le is the smallest relation R such that $\forall x : N, R\,z\,x$ and $\forall x\,y : N, R\,x\,y \to R\,(S\,x)\,(S\,y)$. It gives us a powerful way to prove lemmas of the form $\forall x\,y : N, le\,x\,y \to R\,x\,y$.

Inductive definitions can be defined with parameters. For instance the reflexive-transitive closure of a binary relation R on a type A can be defined as an inductive definition RT with A and R as parameters:

```
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.
```

We have three constructors for this definition: *RTrefl* states that the reflexive-transitive closure is reflexive, *RTR* says it contains R and *RTtran* that it is transitive. For instance, we can prove that the reflexive-transitive closure of the successor relation on N is equivalent to the previously defined *le* relation:

```
∀ x y, le x y ↔ RT N (fun x y ⇒y=S x)
```

Inductive definitions are also used to encode logical operations like absurdity (a definition with no constructor) existential quantification or equality.

```
Inductive False : Prop := .
Inductive ex A (P:A→Prop) : Prop := exists : ∀ x, P x → ex A P.
Inductive eq A (x:A) : A → Prop := eqrefl : eq x x.
```

In first-order logic, when axioms are introduced to form a theory, there is always a risk that it has no model, and consequently everything can be proven. This cannot happen with inductive definitions: there are syntactic restrictions on the type of constructors that ensures the existence of a model.

General rules. The general pattern for declaring an inductive definition is

```
Inductive I pars : Ar := ...
  | c : Π(x1 : A1)..(xn : An), I pars u1..up
  | ...
```

Coq allows the declaration of mutually inductive definitions but, for the sake of simplicity, we shall not give the details here. We introduce some terminology

– $pars$ are called the *parameters* of the inductive definition and will be the same for all definitions;
– Ar is called the *arity*;
– u_i is an *index*;
– $\Pi(x_1 : A_1)..(x_n : A_n), I\, pars\, u_1..u_p$ 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:

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

$$(I : \Pi pars, Ar), pars \vdash C : s$$

- if s is predicative (not Prop) then the condition above on C requires the type of arguments of constructors to be in the same universe: for all i, $A_i : s$ or $A_i : \mathsf{Prop}$
- if s is Prop, we distinguish between *predicative* definitions where $A_i : \mathsf{Prop}$ for all i and *impredicative* definitions otherwise, meaning there is at least one i such that A_i has type Type_i and not Prop.

There is also a positivity condition: occurrences of I 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 does not occur in A_i
– simple case: $A_i = I\, t_1 \dots t_p$ (not necessarily the same parameters), $I \notin t_k$
– functional case: $A_i = \Pi z : B_1, B_2$ with $I \notin B_1$ and I strictly positive in B_2
– nested case: $A_i = J\, t_1 \dots t_q$ with J another inductive definition with parameters $X_1 \dots X_r$. When $t_1 \dots t_r$ are substituted for $X_1 \dots X_r$ in the types of constructors of J, the strict positivity condition should still be satisfied. We also need $I \notin t_k$ for $r < k \leq q$.

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 generalised with respect to the parameters.

$$I : \Pi pars, Ar$$

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:

$$c \equiv \mathtt{Constr}(i, I) : \Pi pars, C$$

Elimination rules use 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):

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

Reduction rule. The reduction rule (called ι) 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).

Examples. This unique rule covers many different situations. In Coq high-level language that we shall use in the examples, we can generally omit the information $\mathbf{as}\, x$, $\mathbf{in}\, I\ \_\ y_1 \ldots y_p$ and $\mathbf{return}\, P$.

The **match** construction can be used to define a function by pattern-matching like the predecessor function.

```
Definition pred (x:N) : N := match x with z ⇒ z | S n ⇒ n end.
```

in this case the return predicate is just $P \stackrel{\text{def}}{=} \texttt{N}$. The same match rule can be used to reason by case analysis on integer and prove the following principle:

$$\forall P : N \to \mathsf{Prop}, P\, z \to (\forall x, P\,(S\, x)) \to (\forall x, P\, x)$$

Given P, $Hz : P\, z$ and $HS : \forall x, P\,(S\, x)$ the following `case` has type $\forall x, P\, x$

```
Definition case (x:N) : P x := match x with z ⇒ Hz | S n ⇒ HS n end.
```

In this example the return predicate is just $P\, n$ and the two branches have two different types: $P\, z$ for the first one and $P\,(S\, n)$ for the second.

The two previous constructions are available in most formalisms. One specificity of CIC is to allow also the definition of new types by pattern-matching. An example can be found when we want to study the semantic of a program and model an environment mapping variables to values. We can represent variables by natural numbers and an environment by a function of type $N \to T$. However with this simple function type, all values should be in the same type. Assume we know a more precise type for some of the variables (the variable 0 as type N and the variable 1 is $N \to N$, other variables are in a default type T), we may want to capture this in the type of environments. We can do it with a construction similar to *pred* but returning a Type as the result instead of a number:

```
Definition env (x:N) : Type
   := match x with z ⇒ N | S z ⇒ N → N | S (S x) ⇒ T end.
```

In this example we use a more elaborate pattern-matching construction that Coq compiles into the primitive forms (using two nested simple pattern-matching). The reduction rule gives us the following equivalence between expressions:

$$env\, z \simeq N \qquad env\,(S\, z) \simeq N \to N \qquad env\,(S\,(S\, x)) \simeq T$$

We need again to use the **match** construction to build a specific environment, but now each branch as a different type, the return predicate is $env\, x$.

```
Definition e1 (x:N) : env x
   := match x with z ⇒ S (S z) | S z ⇒ (fun x : N ⇒ z) | S (S x) ⇒ t end.
```

The result $S\,(S\, z)$ in the first branch (constructor z) has type N which is equivalent to the expected type $env\, z$. The result $(\mathbf{fun}\, x : N \Rightarrow z)$ in the second branch (constructor $S\, z$) has type $N \to N$ which is equivalent to the expected type $env\,(S\, z)$, the last branch returns a default value t in type T which is equivalent to the expected type $env\,(S\,(S\, x))$.

The **match** operation is also available for inductive relations. For instance, for the equality, it gives us the elimination principle we mentioned before

$$\frac{\Gamma \vdash e : t = u \qquad \Gamma, x : A \vdash B : \mathsf{Prop} \qquad \Gamma \vdash f : B[t/x]}{\Gamma \vdash \mathbf{match}\, e\, \mathbf{in}\, \_ = x\, \mathbf{return}\, B\, \mathbf{with}\, \mathit{eqrefl} \Rightarrow f\, \mathbf{end} : B[u/x]}$$

Actually the **match** rule gives us a stronger rule, called a dependent elimination, which allows to replace (in certain contexts) a proof e of $t = u$ by $\mathit{eqrefl}\, t$.

If we take the definition of *le*, the **match** operation will be useful to prove a property like $le\,(S\,x)\,z \to \bot$. Intuitively this is because no constructor can give a proof of an instance of $le\,(S\,x)\,z$. Building the proof term requires to define an invariant relation $inv\,x\,y$ that will be true for all x, y such that $le\,x\,y$ but will be false for $(S\,x)\,z$. It is easily done using **match** to define a proposition which is $\bot$ for $(S\,x)\,z$ and $\top$ for all the other values. We use Coq high-level syntax.

```
Definition inv (x y:N) : Prop :=
  match x,y with S _, z → ⊥ | _,_ ⇒ ⊤ end.
```

We then can build the proof (the term I is a trivial proof of $\top$)

```
Definition leinv n (H:le (S n) z) : ⊥ :=
  match H in le x y return inv x y with
   | lez x ⇒ I | leS x y p ⇒ I end.
```

Type-checking conditions. 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...

Fixpoints. 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`):`$\Pi(x_{n+1} : A_{n+1})\ldots$ $(x_m : A_m)$`,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 $(f : \Pi(x_1 : A_1)\ldots(x_n : A_n), B)$ 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 Properties

3.1 Proof-Theoretical Properties

The main property of the system is decidability of type-checking: we need to be able to say that a proof is correct. It requires the relation $A \preceq B$ to be decidable, which is done by showing the system is strongly normalizing with respect to the computation rules. Another important property is to prove that any closed term with type an inductive definition will reduce to a value starting with one of the constructors of the inductive definition. Consistency can be derived as a consequence because they cannot be a proof without hypothesis of $\bot$ which has no constructor.

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 [3] formalised and proved meta-theoretical properties (including typing decidability assuming normalisation) of a Calculus of Constructions with Inductive Definitions.

3.2 Pragmatic Properties

Dependent types. The type system of CIC allows types to depend on objects in many different ways. We have seen propositions parameterised by objects either defined inductively (*le*) or in a computational way using pattern-matching and fixpoints (*inv*).

It is also possible to mix computational and logical parts. For instance one can build the type of even numbers by defining a predicate `even` and then introducing the type

`Inductive Ne : Type := Nec :` $\Pi n : N,$ `even n →  Nev.`

An object in the type *Ne* will be a pair containing a natural number plus a proof that this number is even. This definition looks like the one for the existential quantifier given in section 2 except that it is a type and not a proposition and consequently it is possible to define a projection function of type $Ne \to N$.

Another possibility to add logical information inside a type is to associate an index to the type declaration representing this information. A classical example is the one of vectors of length n.

```
Inductive vec (A:Type) : N → Type
   := v0 : vec A z | vS : A → vec A n → vec A (S n).
```

We could also associate a predicate to a list describing the set of its elements, we can even use this predicate to ensure that there are no duplicate elements in the list.

`Inductive L (A:Type) : (`*set* `A) →  Type`
$\quad$ `:= L0 : vec` $\emptyset$ `| L1 :` $\Pi(P : \mathit{set}\, A)\ (x : A),\ (x \notin P) \to$ `L A P →  L A (`$P \cup \{x\}$`).`

However manipulating terms in these dependent types might become tricky because the system will generally see the types $vec\, A\, n$ and $vec\, A\, m$ as different, even when n and m can be proven equal if they are not identical with respect to CIC internal equivalence.

Declarative specifications. Inductive definitions of relations can be used for a declarative style of specifications. They 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.

Let us take an example. Assume we want to study a Post correspondence problem given by the three pairs of words on the alphabet $\{a, b\}$: $P \stackrel{\text{def}}{=} \{(a, baa); (ab, aa); (bba, bb)\}$ (example taken from Wikipedia).

We first introduce a data-type in Coq to encode words on the alphabet. We have three constructors: one for the empty word and two unary constructors to add the letter a (resp. b) in front of a word.

```
Inductive word : Type := emp | a : word → word | b : word → word.
```

An inductive relation is used to define a binary relation between words called `post`: two words u and v are related is there exists a sequence of indexes $i_1, \ldots, i_k$ such that $u = u_{i_1} \ldots u_{i_k}$ and $v = v_{i_1} \ldots v_{i_k}$ and $(u_{i_j}, v_{i_j}) \in P$. The constructors of the inductive definition correspond to the basic case where the sequence (and consequently the words u and v) is empty plus one constructor for each pair of words in P.

```
Inductive post : word → word → Prop :=
    start : post emp emp
  | R1 : ∀ x y, post x y → post (a x) (b (a (a y)))
  | R2 : ∀ x y, post x y → post (a (b x)) (a (a y))
  | R3 : ∀ x y, post x y → post (b (b (a x))) (b (b y)).
```

Now a solution is given by a word x which is not empty and such that $(post\, x\, x)$ is provable. A non-empty word can be written ax or bx. One defines the proposition which states that there exists a solution (either starting with a or with b).

```
Inductive sol : Prop :=
  sola : ∀ x, post (a x) (a x) → sol
 |solb : ∀ x, post (b x) (b x) → sol.
```

Now finding a solution is just finding a proof of *sol*. It can be done using automatic tactics. First we declare the constructors of the two definitions *post* and *sol* in the Hint database.

```
Hint Constructors post.
Hint Constructors sol.
```

We can now solve the goal using an automatic tactic which tries to apply the constructors using unification and backtracking:

```
Lemma ok : sol.
eauto 6.
Defined.
```

The **Defined** command at the end builds the proof term which is type-checked by the kernel of Coq verifying that it is indeed a valid term of CIC. The term can be printed

```
ok =
solb (b (a (a (b (b (b (a (a emp))))))))
  (R3 (a (b (b (b (a (a emp)))))) (a (a (b (b (b (a (a emp))))))))
     (R2 (b (b (a (a emp)))) (b (b (b (a (a emp))))))
        (R3 (a emp) (b (a (a emp))) (R1 emp emp start))))
     : sol
```

We see that the proof contains both the witness word, in this case $u \stackrel{\text{def}}{=} bbaabbbaa$ (the first b comes from the fact that the constructor *solb* is used), and a proof of $post\, u\, u$ where we find the sequence 3231 corresponding to the decomposition.

3.3 Computation and Reflection

Inductive definitions help represent data-structures without extra encoding and provide primitives to define recursive functions on these data. The Coq language contains a mini ML sub-language (with no side effect and only terminating functions). It is convenient for formalising complex programs which can then be proven, the most impressive example being the CompCert project of an optimising compiler for C programs [17]. Computation is part of the Coq kernel, many efforts have been made to make it more efficient using compiler technologies.

We can use the underlying language to implement inside Coq various decision procedures. For instance we can have a concrete data structures R to represent symbolic ring expressions with variables, constants, and ring operations. The ring equality eqR can be defined using an inductive definition. Then a simplification function $simpl$ of type $R \to R$ can be defined inside Coq and proven correct with respect to equality $\forall x : R, eqR\, x\, (simpl\, x)$.

We can then 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. If we have in Coq another type C with some operations which have a Ring structure, then it is easy to define recursively a function val of type $R \to C$ which interprets a symbolic expression as an element of C, given an environment which maps variables to elements of C. We have to prove that $\forall (x\, y : R)(\rho : X \to C), eqR\, x\, y \to val\, \rho\, x = val\, \rho\, y$, which is just the justification that our structure C is indeed a ring.

If we put the decision procedure on R and the C interpretation together we can build a proof of

$$\forall (x\, y : R)(\rho : X \to C), val\, \rho\, (simpl\, x) = val\, \rho\, (simpl\, y) \to val\, \rho\, x = val\, \rho\, y$$

This lemma states the (partial)-correctness of the procedure. It is proven once but can be instantiated on many different problems.

Let us illustrate that on our ring example. We want to prove a goal of the form $t = u$ between two Coq expressions in the concrete type C. It might require many rewriting involving associativity, commutativity, distributivity resulting is very large proof terms. Instead we take a detour via our symbolic expressions in R and then use computation. We first write a tactic which guesses by pattern-matching two symbolic expressions p and q (closed terms in R) and an environment ρ such that $val\, \rho\, p \simeq t$ and $val\, \rho\, q \simeq u$. There is always at least one trivial solution with p a variable and $\rho = \{x \mapsto t\}$ for the environment. But of course we want to capture in R as much structure as possible. Now the expressions $simpl\, p$ and $simpl\, q$ can be computed. If they happen to be the same, then $val\, \rho\, (simpl\, x)$ and $val\, \rho\, (simpl\, y)$ are the same Coq term v and so reflexivity justifies that $val\, \rho\, (simpl\, x) = val\, \rho\, (simpl\, y)$. Using the correctness of the procedure we get a proof of $t = u$ that may require a lot of computations but which corresponds to a very simple proof term $correct\, p\, q\, \rho\, (eqrefl\, v)$.

This principle can be used for complex procedures but also for simple reasoning steps. The popular Ssreflect [16] (for small scale reflection) environment (including a tactic language and libraries) which has been successfully used for formalising in Coq the four colour theorem and the Feit-Thompson theorem uses intensively this computational capability of the Coq system mainly on the type of booleans.

4 Proof Applications

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

Many interesting proofs combine advanced algorithms and non-trivial mathematics like the proof of the four-colour theorem by Gonthier & Werner at INRIA and Microsoft-Research [15], a primality checker using Pocklington and Elliptic Curve Certificates developed by Théry et al. at INRIA [25] and the proof of a Wave Equation Resolution Scheme by Boldo et al. [7]. Coq can also be used to certify the output of external theorem provers like in the work on termination tools by Contejean and others [9], or the certification of traces issued from SAT & SMT solvers done by Grégoire and others [2]. Coq is also a good framework for formalising programming environments: the Gemalto and Trusted Logic companies obtained the highest level of certification (common criteria EAL 7) for their formalisation of the security properties of the JavaCard platform [8]; as mentioned earlier Leroy and others developed in Coq a certified optimising compiler for C (Leroy et al.) [17]. Barthe and others used Coq to develop Certicrypt, an environment of formal proofs for computational cryptography [4]. G. Morrisett and others also developed on top of Coq the Ynot library for proving imperative programs using separation logic [19]. These represent typical examples of what can be achieved using Coq.

5 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 (for instance [1]) 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 has been 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 [26] is an attempt to solve this problem. It includes a notion of generalised inductive definitions where the equality definition is included in the declaration, making the definition of quotient types more direct.

6 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. It is integrated in a proof environment (the Coq system) which provides many other functionalities like modules, libraries, notations, type inference, tactics which are essential for a practical use of the formalism. We refer the interested reader to more extensive descriptions of the system like [5, 21].

References

1. Andreas Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006.
2. 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.
3. Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, 1999.
4. 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`.
5. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer-Verlag, 2004.
6. C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39, 1985.
7. 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`.
8. 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`.
9. 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.

10. Th. Coquand. An analysis of girard's paradox. In *Symposium on Logic in Computer Science*, Cambridge, MA, 1986. IEEE Computer Society Press.
11. Th. Coquand and G. Huet. Constructions: A higher order proof system for mechanizing mathematics. In *EUROCAL'85*, volume 203, Linz, 1985. Springer-Verlag.
12. Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 76(2/3), 1988.
13. 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.
14. P. Dybjer. Inductive families. *Formal Asp. Comput.*, 6(4):440–465, 1994.
15. 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`.
16. G. Gonthier and A. Mahboubi. An introduction to small scale reflection in coq. *J. Formalized Reasoning*, 3(2):95–152, 2010.
17. X. Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.
18. P. Martin-Löf. *Intuitionistic Type Theory*. Studies in Proof Theory. Bibliopolis, 1984.
19. G. Morrisett and al. The Ynot project. `http://ynot.cs.harvard.edu/`.
20. 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.
21. C. Paulin-Mohring. *Tools for Practical Software Verification, LASER 2011 summerschool, Revised Tutorial Lectures*, chapter Introduction to the Coq proof-assistant for practical software verification, pages 45–95. Number 7682 in Lecture Notes in Computer Science. Springer-Verlag, 2012.
22. 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.
23. 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.
24. The Coq Development Team. *The Coq Proof Assistant Reference Manual – Version V8.4*, 2012. http://coq.inria.fr.
25. 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/`.
26. The Univalent Foundations Program. *Homotopy Type Theory Univalent Foundations of Mathematics*. Institute for Advanced Study, 2013. `http://homotopytypetheory.org/book/`.