About termination

- Checking convertibility between types may require computing with recursive functions. So, the combination of non-normalization with dependent types leads to undecidable type checking.
- To enforce decidability of type checking, proof assistants either require recursive functions to be encoded in terms of recursors or allow restricted forms of fixpoint expressions.
- A usual way to ensure termination of fixpoint expressions is to impose syntactical restrictions through a predicate $\mathcal{G}_f$ on untyped terms. This predicate enforces termination by constraining all recursive calls to be applied to terms structurally smaller than the formal argument of the function.

The restricted typing rule for fixpoint expressions hence becomes:

$$\frac{\Gamma \vdash \mathbb{N}\to\theta : s \qquad \Gamma, f:\mathbb{N}\to\theta \vdash e:\mathbb{N}\to\theta}{\Gamma \vdash (\mathsf{fix}\ f = e):\mathbb{N}\to\theta} \quad \text{if } \mathcal{G}_f(e)$$

On positivity

Unrestricted general recursion permits the definition of non-terminating functions. So does the possibility of declaring non well-founded datatypes

Consider a datatype $\boldsymbol{d}$ defined by a single introduction rule $\mathtt{C} : (d\to\theta)\to d$, where θ may be any type (even the empty type).

Let $\mathsf{app} \equiv \lambda x.\lambda y.\,\mathsf{case}\ x\ \mathsf{of}\ \{\mathtt{C} \Rightarrow \lambda f.\, f\, y\}$
$\mathsf{t} \equiv (\lambda z.\,\mathsf{app}\, z\, z)$
$\Omega \equiv \mathsf{app}\,(\mathtt{C}\,\mathsf{t})\,(\mathtt{C}\,\mathsf{t})$

We have $\mathsf{app} : d\to d\to\theta$
$\mathsf{t} : d\to\theta$
$\Omega : \theta$

However, Ω is a looping term which has no canonical form

$$\Omega \twoheadrightarrow \mathsf{case}\ (\mathtt{C}\,\mathsf{t})\ \mathsf{of}\ \{\mathtt{C} \Rightarrow \lambda f.\, f\,(\mathtt{C}\,\mathsf{t})\} \to (\lambda f.\, f\,(\mathtt{C}\,\mathsf{t}))\,\mathsf{t} \to \mathsf{t}\,(\mathtt{C}\,\mathsf{t}) \to \Omega$$

What enables to construct a non-normalizing term in θ is the negative occurrence of $\boldsymbol{d}$ in the domain of C.

In order to banish non-well-founded elements from the language, proof assistants usually impose a **positivity condition** on the possible forms of the introduction rules of the inductive types.

Example

The higher-order datatype $\mathbb{O}$: Set of ordinal notations has three constructors

$$\begin{array}{lcl} \mathsf{Zero} & : & \mathbb{O} \\ \mathsf{Succ} & : & \mathbb{O}\to\mathbb{O} \\ \mathsf{Lim} & : & (\mathbb{N}\to\mathbb{O})\to\mathbb{O} \end{array}$$

and comes equipped with a recursor $\mathbf{R}_{\mathbb{O}}$ that can be used to define function and prove properties on ordinals

$$\frac{\begin{array}{cc}\Gamma \vdash P:\mathbb{O}\to\mathsf{Type} & \Gamma \vdash a' : \Pi x{:}\mathbb{O}.\, P\,x \to P\,(\mathsf{Succ}\,x) \\ \Gamma \vdash a : P\,\mathsf{Zero} & \Gamma \vdash a'' : \Pi u{:}\mathbb{N}\to\mathbb{O}.\,(\Pi x{:}\mathbb{N}.\, P\,(u\,x))\to P\,(\mathsf{Lim}\,u)\end{array}}{\Gamma \vdash \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a'' : \Pi n{:}\mathbb{O}.\, P\, n}$$

and its reduction rules are

$$\begin{array}{lcl} \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, \mathsf{Zero} & \to & a \\ \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, (\mathsf{Succ}\, x) & \to & a'\, x\, (\mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, x) \\ \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, (\mathsf{Lim}\, u) & \to & a''\, u\, (\lambda n{:}\mathbb{N}.\, \mathbf{R}_{\mathbb{O}}\, P\, a\, a'\, a''\, (u\, n)) \end{array}$$

Calculus of Inductive Constructions

The CIC is the underlying calculus of Coq. It can be described as follows

Calculus of Inductive Constructions

- **Specification:**

$$\begin{array}{lcl} \mathcal{S} & = & \mathsf{Set}, \mathsf{Prop}, \mathsf{Type}_i \quad , i \in \mathrm{N} \\ \mathcal{A} & = & (\mathsf{Set} : \mathsf{Type}_0),\ (\mathsf{Prop} : \mathsf{Type}_0),\ (\mathsf{Type}_i : \mathsf{Type}_{i+1}) \quad , i \in \mathrm{N} \\ \mathcal{R} & = & (\mathsf{Prop},\mathsf{Prop}),\ (\mathsf{Set},\mathsf{Prop}),\ (\mathsf{Type}_i,\mathsf{Prop}),\ (\mathsf{Prop},\mathsf{Set}),\ (\mathsf{Set},\mathsf{Set}),\ (\mathsf{Type}_i,\mathsf{Set}) \\ & & (\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\max(i,j)}) \quad , i,j \in \mathrm{N} \end{array}$$

- **Cumulativity**: $\mathsf{Prop} \subseteq \mathsf{Type}_0$, $\mathsf{Set} \subseteq \mathsf{Type}_0$ and $\mathsf{Type}_i \subseteq \mathsf{Type}_{i+1}$, $i \in \mathrm{N}$.
- **Inductive types** and a restricted form of general recursion.

In the Coq system, the user will never mention explicitly the index i when referring to the universe Type_i . One only writes Type. The system itself generates for each instance of Type a new index for the universe and checks that the constraints between these indexes can be solved.
From the user point of view we consequently have Type : Type

Impredicativity

In **CIC**, thanks to rules $(\text{Type}_i, \text{Prop})$ and $(\text{Type}_i, \text{Set})$, the following judgments are derivable

$$\vdash (\Pi A\!:\!\text{Prop}.\, A \to A) : \text{Prop}$$
$$\vdash (\Pi A\!:\!\text{Set}.\, A \to A) : \text{Set}$$

which means that:

- it is possible to construct a new element of type Prop by quantifying over all elements of type Prop;
- it is possible to construct a new element of type Set by quantifying over all elements of type Set.

These kinds of types are called **impredicative**.

In this case we say Prop and Set are impredicative universes.

Coq version V7 was based in CIC.

Impredicativity (cont.)

Coq version V8 is based in a weaker calculus:

the **Predicative Calculus of Inductive Constructions** (pCIC) .

In **pCIC** the rule $(\text{Type}_i, \text{Set})$ was removed, as a consequence: the universe Set become predicative.

- Within **pCIC** the type $\Pi A\!:\!\text{Set}.\, A \to A$ has now sort Type.
- Prop is the only impredicative universe of **pCIC**.

NOTE: The only possible universes where impredicativity is allowed are the ones at the bottom of the hierarchy. Otherwise the calculus would turn out inconsistent.
This justifies the rules $(\text{Type}_i, \text{Type}_j, \text{Type}_{\max(i,j)}),\ i,j \in \mathbb{N})$

Coq in brief

In the Coq system the well typing of a term depends on an environment which consists in a **global environment** and a **local context**.

- The local context is a sequence of variable declarations, written $x : A$ (A is a type) and "standard" definitions, written $x := t : A$ (i.e., abbreviations for well-formed terms).
- The global environment is list of global declarations and definitions. This includes not only assumptions and "standard" definitions, but also definitions of inductive objects. (The global environment can be set by loading some libraries.)

We frequently use the names constant to describe a globally defined identifier and global variable for a globally declared identifier.

The typing judgments are as follows: $E \,|\, \Gamma \vdash t : A$

Declarations and definitions

The environment combines the contents of initial environment, the loaded libraries, and all the global definitions and declarations made by the user.

Loading modules

`Require Import ZArith.` This command loads the definitions and declarations of module **`ZArith`** which is the standard library for basic relative integer arithmetic.

The Coq system has a block mechanism **`Section`** *`id`*`.` ... **`End`** *`id`*`.` which allows to manipulate the local context (by expanding and contracting it).

Declarations

```
Parameter max_int : Z.            Global variable declaration.

Section Example.

Variables A B : Set.              Local variable declarations.
Variable Q : Prop.
Variables (b:B) (P : A->Prop).
```

Declarations and definitions (cont.)

Definitions

```
Definition min_int := (1 - max_int)%Z.
```
Global definition.

```
Definition FB := B -> B.
```
Local definition.

Proof-terms

```
Lemma trivial : forall x:A, P x -> P x.
intros x H.
exact H.
Qed.
```

- Using tactics a term of type `forall x:A, P x -> P x` has been created.
- Using `Qed` the identifier `trivial` is defined as this proof-term and add to the global environment.

Syntax

$\lambda x{:}A.\, \lambda y{:}A \to B.\; y\, x$	`fun (x:A) (y:A->B) => y x`
$\Pi x{:}A.\; P\,x \to P\,x$	`forall x:A, P x -> P x`

Inductive types

```
Inductive nat :Set :=
  | O : nat
  | S : nat -> nat.
```

This definition yields:
- constructors `O` and `S`
- recursors `nat_ind`, `nat_rec`, `nat_rect`

General recursion + case analysis

```
Fixpoint double (n:nat) :nat :=
  match n with
    | O      => O
    | (S x)  => S (S (double x))
  end.
```

Note that the recursive call is "smaller".

Inductive types

Example:

```
Inductive nat :Set := O : nat
                    | S : nat -> nat.
```

The declaration of this inductive type introduces in the global environment not only the constructors `O` and `S`, but also the recursors: `nat_rect`, `nat_ind` and `nat_rec`

Recursor

```
Coq < Check nat_rect.
nat_rect
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Proof-by-induction scheme

```
Coq < Print nat_ind.
nat_ind = fun P : nat -> Prop => nat_rect P
   : forall P : nat -> Prop,
     P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```

Primitive recursor scheme

```
Coq < Print nat_rec.
nat_rec = fun P : nat -> Set => nat_rect P
   : forall P : nat -> Set,
     P 0 -> (forall n : nat, P n -> P (S n)) -> forall n : nat, P n
```


Computations

Computations are performed as series of **reductions**. The `Eval` command computes the normal form of a term with respect to some reduction rules (and using some reduction strategy: `cbv` or `lazy`).

- **β-reduction** $\quad (\lambda x : A.\, M)\, N \;\to_\beta\; M[x := N]$
- **δ-reduction** , for unfolding definitions $\quad D \;\to_\delta\; M \quad \text{if } (D := M) \in E \,|\, \Gamma$
- **ι-reduction** , for primitive recursion rules, general recursion and case analysis
- **ζ-reduction** , for local definitions $\quad \texttt{let } x := a \texttt{ in } b \;\to_\zeta\; b[x := a]$

Note that the conversion rule is

$$\frac{E\,|\,\Gamma \;\vdash\; M : A \qquad E\,|\,\Gamma \;\vdash\; B : s}{E\,|\,\Gamma \;\vdash\; M : B} \quad \text{if } A =_{\beta\iota\delta\zeta} B$$

The cumulativity property within the universe hierarchy leads to a notion of order between types, written $E\,|\,\Gamma \;\vdash\; A \leq_{\beta\iota\delta\zeta} B$, which replaces the side condition in the conversion rule. A precise description of this relation can be found in the Coq reference manual.

Implicit syntax

The symbol _ can be used to replace a function argument when the context makes it possible to determine automatically the value of this argument. When handling terms, the Coq system simply replaces each _ by the appropriate value.

```
Definition compose : forall A B C : Set, (A->B) -> (B->C) -> A -> C
      := fun A B C f g x => g (f x).
```

```
Coq < Check (fun (A:Set) (f:nat->A) => compose _ _ _ double f).
fun (A : Set) (f : nat -> A) => compose nat nat A double f
     : forall A : Set, (nat -> A) -> nat -> A
```

The **implicit arguments mechanism** makes possible to avoid _ in Coq expressions. It is necessary to describe in advance the arguments that should be inferred from the other arguments of a function f or from the context, when writing an application of f these arguments must be omitted.

Implicit syntax (cont.)

```
Implicit Arguments compose [A B C].

Coq < Check (compose double S).
compose double S
   : nat -> nat
```

If the Coq system cannot infer the implicit arguments it is possible to give them explicitly.

```
Coq < Check (compose (C:=nat) double).
compose (C:=nat) double
   : (nat -> nat) -> nat -> nat
```

The Coq system also provides a working mode where the arguments that could be inferred are automatically determined and declared as implicit arguments when a function is defined.

```
Set Implicit Arguments.
```

To deactivate this mode:

```
Unset Implicit Arguments.
```


Lists

An example of a parametric inductive type: the type of lists over a type A.

```
Inductive list (A : Type) : Type :=
  | nil : list A
  | cons : A -> list A -> list A.
```

In this definition, A is a general parameter, global to both constructors. This kind of definition allows us to build a whole family of inductive types, indexed over the sort `Type`.

The recursor for lists

```
Coq < Check list_rect.
list_rect
     : forall (A : Type) (P : list A -> Type),
       P nil ->
       (forall (a : A) (l : list A), P l -> P (cons a l)) ->
       forall l : list A, P l
```


Vectors of length n over A

```
Inductive vector (A : Type) : nat -> Type :=
  | Vnil : vector A O
  | Vcons : A -> forall n : nat, vector A n -> vector A (S n).
```

Remark the difference between the two parameters A and n:
- A is general parameter, global to all the introduction rules;
- n is an index, which is instantiated differently in the introduction rules.

The type of constructor `Vcons` is a dependent type.

```
Variables b1 b2 : B.

Coq < Check (Vcons _ b1 _ (Vcons _ b2 _ (Vnil _))).
Vcons B b1 1 (Vcons B b2 0 (Vnil B))
     : vector B 2
```

The recursor for vectors

```
Coq < Check vector_rect.
vector_rect
     : forall (A : Type) (P : forall n : nat, vector A n -> Type),
       P 0 (Vnil A) ->
       (forall (a : A) (n : nat) (v : vector A n),
       P n v -> P (S n) (Vcons A a n v)) ->
       forall (n : nat) (v : vector A n), P n v
```

Equality

In Coq, the propositional equality between two inhabitants a and b of the same type A is introduced as a family of recursive predicates "to be equal to a", parameterized by both a and its type A. This family of types has only one introduction rule, which corresponds to reflexivity.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : (eq A x x).
```

The induction principle of eq is very close to the Leibniz's equality but not exactly the same.

```
Coq < Check eq_ind.
eq_ind
     : forall (A : Type) (x : A) (P : A -> Prop),
       P x -> forall y : A, x = y -> P y
```

Notice that Coq system uses the syntax "a = b" is an abbreviation for "eq a b", and that the parameter A is implicit, as it can be inferred from a.

```
Inductive eq (A : Type) (x : A) : A -> Prop :=
  | refl_equal : x = x.
```


Relations as inductive types

Some relations can be introduced as an inductive family of propositions. For instance, the order $n \leq m$ on natural numbers is defined as follows in the standard library:

```
Inductive le (n:nat) : nat -> Prop :=
  | le_n : le n n
  | le_S : forall m, le n m -> le n (S m).
```

- Notice that in this definition n is a general parameter, while the second argument of le is an index. This definition introduces the binary relation $n \leq m$ as the family of unary predicates "to be greater or equal than a given n", parameterized by n.
- The Coq system provides a syntactic convention, so that "le x y" can be written "le x <= y".
- The introduction rules of this type can be seen as rules for proving that a given integer n is less or equal than another one. In fact, an object of type $n \leq m$ is nothing but a proof built up using the constructors **le_n** and **le_S**.

Sigma types

The concept of Σ-type is implemented in Coq by the following inductive type.

```
Inductive sig (A : Type) (P : A -> Prop) : Type :=
  | exist : forall x : A, P x -> sig A P.

Implicit Arguments sig [A].
```

- Note that this inductive type can be used to build a specification, combining a datatype and a predicate over this type, thus creating "the type of data that satisfies the predicate". Intuitively, the type one obtains represents a subset of the initial type.
- The Coq system provides a syntactical convention for this inductive type. For instance, assume we have a predicate **prime : nat -> Prop** in the environment. The expression **sig prime** (notice the implicit argument) can be written **{x:nat | prime x}**.
- A certified value of this type should contain a **computation** component that says how to obtain a value n and a **certificate**, a proof that is n a prime.

Logical connectives in Coq

In the Coq System, most logical connectives are represented as inductive types, except for ⊃ and ∀ which are directly represented by → and Π-types, and negation which is defined as the implication of the absurd.

```
Definition not := fun A : Prop => A -> False.
```
~ is pretty printing for **not**

```
Inductive False : Prop := .
```

```
Inductive True : Prop := I : True.
```

```
Inductive and (A : Prop) (B : Prop) : Prop :=
  | conj : A -> B -> (and A B).
```
/\ is pretty printing for **and**

```
Inductive or (A : Prop) (B : Prop) : Prop :=
  | or_introl : A -> (or A B)
  | or_intror : B -> (or A B).
```
\/ is pretty printing for **or**

```
Inductive ex (A : Type) (P : A -> Prop) : Prop :=
  | ex_intro : forall x : A, P x -> ex A P.
```

The constructors are the introduction rules.
The induction principle gives the elimination rules.