spec_float, and the injection Prim2SF:

```
Print spec_float.
    Variant spec_float : Set :=
        S754_zero : bool -> spec_float
      | S754_infinity : bool -> spec_float
      | S754_nan : spec_float
      | S754_finite : bool -> positive -> Z -> spec_float

    Arguments S754_zero _%bool_scope
    Arguments S754_infinity _%bool_scope
    Arguments S754_finite _%bool_scope _%positive_scope _%Z_scope

Check Prim2SF.
    Prim2SF
         : float -> spec_float

Check mul_spec.
    mul_spec
         : forall x y : float, Prim2SF (x * y) = SF64mul (Prim2SF x) (Prim2SF y)
```

For more details on the available definitions and lemmas, see the online documentation of the Floats library.

4.3.3 Users' contributions

Numerous users' contributions have been collected and are available at URL http://coq.inria.fr/opam/www/. On this web page, you have a list of all contributions with informations (author, institution, quick description, etc.) and the possibility to download them one by one. You will also find informations on how to submit a new contribution.

4.4 Calculus of Inductive Constructions

The underlying formal language of Coq is a *Calculus of Inductive Constructions* (Cic) whose inference rules are presented in this chapter. The history of this formalism as well as pointers to related work are provided in a separate chapter; see *Credits*.

4.4.1 The terms

The expressions of the Cic are *terms* and all terms have a *type*. There are types for functions (or programs), there are atomic types (especially datatypes)... but also types for proofs and types for the types themselves. Especially, any object handled in the formalism must belong to a type. For instance, universal quantification is relative to a type and takes the form "*for all* x *of type* T, P". The expression "x *of type* T" is written "$x : T$". Informally, "$x : T$" can be thought as "x *belongs to* T".

The types of types are *sorts*. Types and sorts are themselves terms so that terms, types and sorts are all components of a common syntactic language of terms which is described in Section *Terms* but, first, we describe sorts.

Sorts

All sorts have a type and there is an infinite well-founded typing hierarchy of sorts whose base sorts are SProp, Prop and Set.

The sort Prop intends to be the type of logical propositions. If M is a logical proposition then it denotes the class of terms representing proofs of M. An object m belonging to M witnesses the fact that M is provable. An object of type Prop is called a proposition.

The sort SProp is like Prop but the propositions in SProp are known to have irrelevant proofs (all proofs are equal). Objects of type SProp are called strict propositions. SProp is rejected except when using the compiler option `-allow-sprop`. See *SProp (proof irrelevant propositions)* for information about using SProp, and *[GCST19]* for meta theoretical considerations.

The sort Set intends to be the type of small sets. This includes data types such as booleans and naturals, but also products, subsets, and function types over these data types.

SProp, Prop and Set themselves can be manipulated as ordinary terms. Consequently they also have a type. Because assuming simply that Set has type Set leads to an inconsistent theory *[Coq86]*, the language of Cic has infinitely many sorts. There are, in addition to the base sorts, a hierarchy of universes Type(i) for any integer $i \geq 1$.

Like Set, all of the sorts Type(i) contain small sets such as booleans, natural numbers, as well as products, subsets and function types over small sets. But, unlike Set, they also contain large sets, namely the sorts Set and Type(j) for $j < i$, and all products, subsets and function types over these sorts.

Formally, we call $\mathcal{S}$ the set of sorts which is defined by:

$$\mathcal{S} \equiv \{\mathsf{SProp}, \mathsf{Prop}, \mathsf{Set}, \mathsf{Type}(i) \mid i \ \in \mathbb{N}\}$$

Their properties, such as: Prop : Type(1), Set : Type(1), and Type(i) : Type($i+1$), are defined in Section *Subtyping rules*.

The user does not have to 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. We shall make precise in the typing rules the constraints between the indices.

Implementation issues In practice, the Type hierarchy is implemented using *algebraic universes*. An algebraic universe u is either a variable (a qualified identifier with a number) or a successor of an algebraic universe (an expression $u+1$), or an upper bound of algebraic universes (an expression $\max(u_1, ..., u_n)$), or the base universe (the expression 0) which corresponds, in the arity of template polymorphic inductive types (see Section *Well-formed inductive definitions*), to the predicative sort Set. A graph of constraints between the universe variables is maintained globally. To ensure the existence of a mapping of the universes to the positive integers, the graph of constraints must remain acyclic. Typing expressions that violate the acyclicity of the graph of constraints results in a Universe inconsistency error.

See also:

Section *Printing universes*.

Terms

Terms are built from sorts, variables, constants, abstractions, applications, local definitions, and products. From a syntactic point of view, types cannot be distinguished from terms, except that they cannot start by an abstraction or a constructor. More precisely the language of the *Calculus of Inductive Constructions* is built from the following rules.

1. the sorts SProp, Prop, Set, Type(i) are terms.
2. variables, hereafter ranged over by letters x, y, etc., are terms
3. constants, hereafter ranged over by letters c, d, etc., are terms.

4. if x is a variable and T, U are terms then $\forall x : T,\ U$ (`forall x:T, U` in Coq concrete syntax) is a term. If x occurs in U, $\forall x : T,\ U$ reads as "for all x of type T, U". As U depends on x, one says that $\forall x : T,\ U$ is a *dependent product*. If x does not occur in U then $\forall x : T,\ U$ reads as "if T then U". A *non dependent product* can be written: $T \to U$.

5. if x is a variable and T, u are terms then $\lambda x : T.\ u$ (`fun x:T => u` in Coq concrete syntax) is a term. This is a notation for the λ-abstraction of λ-calculus *[Bar81]*. The term $\lambda x : T.\ u$ is a function which maps elements of T to the expression u.

6. if t and u are terms then $(t\ u)$ is a term (`t u` in Coq concrete syntax). The term $(t\ u)$ reads as "t applied to u".

7. if x is a variable, and t, T and u are terms then $\mathsf{let}\ x := t : T\ \mathsf{in}\ u$ is a term which denotes the term u where the variable x is locally bound to t of type T. This stands for the common "let-in" construction of functional programs such as ML or Scheme.

Free variables. The notion of free variables is defined as usual. In the expressions $\lambda x : T.\ U$ and $\forall x : T,\ U$ the occurrences of x in U are bound.

Substitution. The notion of substituting a term t to free occurrences of a variable x in a term u is defined as usual. The resulting term is written $u\{x/t\}$.

The logical vs programming readings. The constructions of the Cic can be used to express both logical and programming notions, accordingly to the Curry-Howard correspondence between proofs and programs, and between propositions and types *[CFC58][How80][dB72]*.

For instance, let us assume that nat is the type of natural numbers with zero element written 0 and that True is the always true proposition. Then $\to$ is used both to denote nat $\to$ nat which is the type of functions from nat to nat, to denote True$\to$True which is an implicative proposition, to denote nat $\to$ Prop which is the type of unary predicates over the natural numbers, etc.

Let us assume that `mult` is a function of type nat $\to$ nat $\to$ nat and `eqnat` a predicate of type nat $\to$ nat $\to$ Prop. The λ-abstraction can serve to build "ordinary" functions as in $\lambda x : \mathsf{nat}.\ (\mathsf{mult}\ x\ x)$ (i.e. `fun x:nat => mult x x` in Coq notation) but may build also predicates over the natural numbers. For instance $\lambda x : \mathsf{nat}.\ (\mathsf{eqnat}\ x\ 0)$ (i.e. `fun x:nat => eqnat x 0` in Coq notation) will represent the predicate of one variable x which asserts the equality of x with 0. This predicate has type nat $\to$ Prop and it can be applied to any expression of type nat, say t, to give an object $P\ t$ of type Prop, namely a proposition.

Furthermore `forall x:nat, P x` will represent the type of functions which associate to each natural number n an object of type $(P\ n)$ and consequently represent the type of proofs of the formula "$\forall x.\ P(x)$".

4.4.2 Typing rules

As objects of type theory, terms are subjected to *type discipline*. The well typing of a term depends on a global environment and a local context.

Local context. A *local context* is an ordered list of *local declarations* of names which we call *variables*. The declaration of some variable x is either a *local assumption*, written $x : T$ (T is a type) or a *local definition*, written $x := t : T$. We use brackets to write local contexts. A typical example is $[x : T;\ y := u : U;\ z : V]$. Notice that the variables declared in a local context must be distinct. If Γ is a local context that declares some x, we write $x \in \Gamma$. By writing $(x : T) \in \Gamma$ we mean that either $x : T$ is an assumption in Γ or that there exists some t such that $x := t : T$ is a definition in Γ. If Γ defines some $x := t : T$, we also write $(x := t : T) \in \Gamma$. For the rest of the chapter, $\Gamma :: (y : T)$ denotes the local context Γ enriched with the local assumption $y : T$. Similarly, $\Gamma :: (y := t : T)$ denotes the local context Γ enriched with the local definition $(y := t : T)$. The notation [] denotes the empty local context. By $\Gamma_1; \Gamma_2$ we mean concatenation of the local context Γ_1 and the local context Γ_2.

Global environment. A *global environment* is an ordered list of *global declarations*. Global declarations are either *global assumptions* or *global definitions*, but also declarations of inductive objects. Inductive objects themselves declare both inductive or coinductive types and constructors (see Section *Inductive Definitions*).

A *global assumption* will be represented in the global environment as $(c : T)$ which assumes the name c to be of some type T. A *global definition* will be represented in the global environment as $c := t : T$ which defines the name c to have value t and type T. We shall call such names *constants*. For the rest of the chapter, the $E;\ c : T$ denotes the global environment E enriched with the global assumption $c : T$. Similarly, $E;\ c := t : T$ denotes the global environment E enriched with the global definition $(c := t : T)$.

The rules for inductive definitions (see Section *Inductive Definitions*) have to be considered as assumption rules to which the following definitions apply: if the name c is declared in E, we write $c \in E$ and if $c : T$ or $c := t : T$ is declared in E, we write $(c : T) \in E$.

Typing rules. In the following, we define simultaneously two judgments. The first one $E[\Gamma] \vdash t : T$ means the term t is well-typed and has type T in the global environment E and local context Γ. The second judgment $\mathcal{WF}(E)[\Gamma]$ means that the global environment E is well-formed and the local context Γ is a valid local context in this global environment.

A term t is well typed in a global environment E iff there exists a local context Γ and a term T such that the judgment $E[\Gamma] \vdash t : T$ can be derived from the following rules.

W-Empty

$$\frac{}{\mathcal{WF}([])[]}$$

W-Local-Assum

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x : T)]}$$

W-Local-Def

$$\frac{E[\Gamma] \vdash t : T \qquad x \notin \Gamma}{\mathcal{WF}(E)[\Gamma :: (x := t : T)]}$$

W-Global-Assum

$$\frac{E[] \vdash T : s \qquad s \in \mathcal{S} \qquad c \notin E}{\mathcal{WF}(E;\ c : T)[]}$$

W-Global-Def

$$\frac{E[] \vdash t : T \qquad c \notin E}{\mathcal{WF}(E;\ c := t : T)[]}$$

Ax-SProp

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{SProp} : \mathsf{Type}(1)}$$

Ax-Prop

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Prop} : \mathsf{Type}(1)}$$

Ax-Set

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Set} : \mathsf{Type}(1)}$$

Ax-Type

$$\frac{\mathcal{WF}(E)[\Gamma]}{E[\Gamma] \vdash \mathsf{Type}(i) : \mathsf{Type}(i+1)}$$

Var

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x : T) \in \Gamma \text{ or } (x := t : T) \in \Gamma \text{ for some } t}{E[\Gamma] \vdash x : T}$$

Const

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c : T) \in E \text{ or } (c := t : T) \in E \text{ for some } t}{E[\Gamma] \vdash c : T}$$

Prod-SProp

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{SProp}}{E[\Gamma] \vdash \forall\ x : T, U : \mathsf{SProp}}$$

Prod-Prop

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Prop}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Prop}}$$

Prod-Set

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Prop}, \mathsf{Set}\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Set}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Set}}$$

Prod-Type

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \{\mathsf{SProp}, \mathsf{Type}i\} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Type}(i)}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Type}(i)}$$

Lam

$$\frac{E[\Gamma] \vdash \forall x : T,\ U : s \qquad E[\Gamma :: (x : T)] \vdash t : U}{E[\Gamma] \vdash \lambda x : T.\ t : \forall x : T,\ U}$$

App

$$\frac{E[\Gamma] \vdash t : \forall x : U,\ T \qquad E[\Gamma] \vdash u : U}{E[\Gamma] \vdash (t\ u) : T\{x/u\}}$$

Let

$$\frac{E[\Gamma] \vdash t : T \qquad E[\Gamma :: (x := t : T)] \vdash u : U}{E[\Gamma] \vdash \mathsf{let}\ x := t : T\ \mathsf{in}\ u : U\{x/t\}}$$

Note: **Prod-Prop** and **Prod-Set** typing-rules make sense if we consider the semantic difference between Prop and Set:

- All values of a type that has a sort Set are extractable.
- No values of a type that has a sort Prop are extractable.

Note: We may have let $x := t : T$ in u well-typed without having $((\lambda x : T.\ u)\ t)$ well-typed (where T is a type of t). This is because the value t associated to x may be used in a conversion rule (see Section *Conversion rules*).

4.4.3 Conversion rules

In Cic, there is an internal reduction mechanism. In particular, it can decide if two programs are *intentionally* equal (one says *convertible*). Convertibility is described in this section.

β-reduction

We want to be able to identify some terms as we can identify the application of a function to a given argument with its result. For instance the identity function over a given type T can be written $\lambda x : T.\ x$. In any global environment E and local context Γ, we want to identify any object a (of type T) with the application $((\lambda x : T.\ x)\ a)$. We define for this a *reduction* (or a *conversion*) rule we call β:

$$E[\Gamma] \vdash ((\lambda x : T.\ t)\ u)\ \rhd_\beta\ t\{x/u\}$$

We say that $t\{x/u\}$ is the β-*contraction* of $((\lambda x : T.\ t)\ u)$ and, conversely, that $((\lambda x : T.\ t)\ u)$ is the β-*expansion* of $t\{x/u\}$.

According to β-reduction, terms of the *Calculus of Inductive Constructions* enjoy some fundamental properties such as confluence, strong normalization, subject reduction. These results are theoretically of great importance but we will not detail them here and refer the interested reader to *[Coq85]*.

ι-reduction

A specific conversion rule is associated to the inductive objects in the global environment. We shall give later on (see Section *Well-formed inductive definitions*) the precise rules but it just says that a destructor applied to an object built from a constructor behaves as expected. This reduction is called ι-reduction and is more precisely studied in *[PM93a][Wer94]*.

δ-reduction

We may have variables defined in local contexts or constants defined in the global environment. It is legal to identify such a reference with its value, that is to expand (or unfold) it into its value. This reduction is called δ-reduction and shows as follows.

Delta-Local

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (x := t : T) \in \Gamma}{E[\Gamma] \vdash x\ \rhd_\Delta\ t}$$

Delta-Global

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad (c := t : T) \in E}{E[\Gamma] \vdash c \ \triangleright_\delta \ t}$$

ζ-reduction

Coq allows also to remove local definitions occurring in terms by replacing the defined variable by its value. The declaration being destroyed, this reduction differs from δ-reduction. It is called ζ-reduction and shows as follows.

Zeta

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad E[\Gamma] \vdash u : U \qquad E[\Gamma :: (x := u : U)] \vdash t : T}{E[\Gamma] \vdash \mathsf{let}\ x := u : U\ \mathsf{in}\ t \ \triangleright_\zeta \ t\{x/u\}}$$

η-expansion

Another important concept is η-expansion. It is legal to identify any term t of functional type $\forall x : T,\ U$ with its so-called η-expansion

$$\lambda x : T.\ (t\ x)$$

for x an arbitrary variable name fresh in t.

Note: We deliberately do not define η-reduction:

$$\lambda x : T.\ (t\ x) \ \not\triangleright_\eta \ t$$

This is because, in general, the type of t need not to be convertible to the type of $\lambda x : T.\ (t\ x)$. E.g., if we take f such that:

$$f \ : \ \forall x : \mathsf{Type}(2),\ \mathsf{Type}(1)$$

then

$$\lambda x : \mathsf{Type}(1).\ (f\ x) \ : \ \forall x : \mathsf{Type}(1),\ \mathsf{Type}(1)$$

We could not allow

$$\lambda x : \mathsf{Type}(1).\ (f\ x) \ \triangleright_\eta \ f$$

because the type of the reduced term $\forall x : \mathsf{Type}(2),\ \mathsf{Type}(1)$ would not be convertible to the type of the original term $\forall x : \mathsf{Type}(1),\ \mathsf{Type}(1)$.

Proof Irrelevance

It is legal to identify any two terms whose common type is a strict proposition $A : \mathsf{SProp}$. Terms in a strict propositions are therefore called *irrelevant*.

Convertibility

Let us write $E[\Gamma] \vdash t \rhd u$ for the contextual closure of the relation t reduces to u in the global environment E and local context Γ with one of the previous reductions β, δ, ι or ζ.

We say that two terms t_1 and t_2 are *$\beta\delta\iota\zeta\eta$-convertible*, or simply *convertible*, or *equivalent*, in the global environment E and local context Γ iff there exist terms u_1 and u_2 such that $E[\Gamma] \vdash t_1 \rhd ... \rhd u_1$ and $E[\Gamma] \vdash t_2 \rhd ... \rhd u_2$ and either u_1 and u_2 are identical up to irrelevant subterms, or they are convertible up to η-expansion, i.e. u_1 is $\lambda x : T.\ u_1'$ and $u_2 x$ is recursively convertible to u_1', or, symmetrically, u_2 is $\lambda x : T.\ u_2'$ and $u_1 x$ is recursively convertible to u_2'. We then write $E[\Gamma] \vdash t_1 =_{\beta\delta\iota\zeta\eta} t_2$.

Apart from this we consider two instances of polymorphic and cumulative (see Chapter *Polymorphic Universes*) inductive types (see below) convertible

$$E[\Gamma] \vdash t\ w_1 ... w_m =_{\beta\delta\iota\zeta\eta} t\ w_1' ... w_m'$$

if we have subtypings (see below) in both directions, i.e.,

$$E[\Gamma] \vdash t\ w_1 ... w_m \leq_{\beta\delta\iota\zeta\eta} t\ w_1' ... w_m'$$

and

$$E[\Gamma] \vdash t\ w_1' ... w_m' \leq_{\beta\delta\iota\zeta\eta} t\ w_1 ... w_m.$$

Furthermore, we consider

$$E[\Gamma] \vdash c\ v_1 ... v_m =_{\beta\delta\iota\zeta\eta} c'\ v_1' ... v_m'$$

convertible if

$$E[\Gamma] \vdash v_i =_{\beta\delta\iota\zeta\eta} v_i'$$

and we have that c and c' are the same constructors of different instances of the same inductive types (differing only in universe levels) such that

$$E[\Gamma] \vdash c\ v_1 ... v_m : t\ w_1 ... w_m$$

and

$$E[\Gamma] \vdash c'\ v_1' ... v_m' : t'\ w_1' ... w_m'$$

and we have

$$E[\Gamma] \vdash t\ w_1 ... w_m =_{\beta\delta\iota\zeta\eta} t\ w_1' ... w_m'.$$

The convertibility relation allows introducing a new typing rule which says that two convertible well-formed types have the same inhabitants.

4.4.4 Subtyping rules

At the moment, we did not take into account one rule between universes which says that any term in a universe of index i is also a term in the universe of index $i + 1$ (this is the *cumulativity* rule of Cic). This property extends the equivalence relation of convertibility into a *subtyping* relation inductively defined by:

1. if $E[\Gamma] \vdash t =_{\beta\delta\iota\zeta\eta} u$ then $E[\Gamma] \vdash t \leq_{\beta\delta\iota\zeta\eta} u$,
2. if $i \leq j$ then $E[\Gamma] \vdash \mathsf{Type}(i) \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(j)$,

3. for any i, $E[\Gamma] \vdash \mathsf{Set} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(i)$,
4. $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Set}$, hence, by transitivity, $E[\Gamma] \vdash \mathsf{Prop} \leq_{\beta\delta\iota\zeta\eta} \mathsf{Type}(i)$, for any i (note: SProp is not related by cumulativity to any other term)
5. if $E[\Gamma] \vdash T =_{\beta\delta\iota\zeta\eta} U$ and $E[\Gamma :: (x : T)] \vdash T' \leq_{\beta\delta\iota\zeta\eta} U'$ then $E[\Gamma] \vdash \forall x : T,\ T' \leq_{\beta\delta\iota\zeta\eta} \forall x : U,\ U'$.
6. if $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$ is a universe polymorphic and cumulative (see Chapter *Polymorphic Universes*) inductive type (see below) and $(t : \forall \Gamma_P, \forall \Gamma_{Arr(t)}, S) \in \Gamma_I$ and $(t' : \forall \Gamma'_P, \forall \Gamma'_{Arr(t)}, S') \in \Gamma_I$ are two different instances of *the same* inductive type (differing only in universe levels) with constructors

$$[c_1 : \forall \Gamma_P, \forall T_{1,1} ... T_{1,n_1},\ t\ v_{1,1} ... v_{1,m};\ ...;\ c_k : \forall \Gamma_P, \forall T_{k,1} ... T_{k,n_k},\ t\ v_{k,1} ... v_{k,m}]$$

and

$$[c_1 : \forall \Gamma'_P, \forall T'_{1,1} ... T'_{1,n_1},\ t'\ v'_{1,1} ... v'_{1,m};\ ...;\ c_k : \forall \Gamma'_P, \forall T'_{k,1} ... T'_{k,n_k},\ t'\ v'_{k,1} ... v'_{k,m}]$$

respectively then

$$E[\Gamma] \vdash t\ w_1 ... w_m \leq_{\beta\delta\iota\zeta\eta} t'\ w'_1 ... w'_m$$

(notice that t and t' are both fully applied, i.e., they have a sort as a type) if

$$E[\Gamma] \vdash w_i =_{\beta\delta\iota\zeta\eta} w'_i$$

for $1 \leq i \leq m$ and we have

$$E[\Gamma] \vdash T_{i,j} \leq_{\beta\delta\iota\zeta\eta} T'_{i,j}$$

and

$$E[\Gamma] \vdash A_i \leq_{\beta\delta\iota\zeta\eta} A'_i$$

where $\Gamma_{Arr(t)} = [a_1 : A_1;\ ...;\ a_l : A_l]$ and $\Gamma'_{Arr(t)} = [a_1 : A'_1;\ ...;\ a_l : A'_l]$.

The conversion rule up to subtyping is now exactly:

Conv

$$\frac{E[\Gamma] \vdash U : s \qquad E[\Gamma] \vdash t : T \qquad E[\Gamma] \vdash T \leq_{\beta\delta\iota\zeta\eta} U}{E[\Gamma] \vdash t : U}$$

Normal form. A term which cannot be any more reduced is said to be in *normal form*. There are several ways (or strategies) to apply the reduction rules. Among them, we have to mention the *head reduction* which will play an important role (see Chapter *Tactics*). Any term t can be written as $\lambda x_1 : T_1. ... \lambda x_k : T_k.\ (t_0\ t_1 ... t_n)$ where t_0 is not an application. We say then that t_0 is the *head of* t. If we assume that t_0 is $\lambda x : T.\ u_0$ then one step of β-head reduction of t is:

$$\lambda x_1 : T_1. ... \lambda x_k : T_k.\ (\lambda x : T.\ u_0\ t_1 ... t_n) \ \triangleright\ \lambda (x_1 : T_1) ... (x_k : T_k).\ (u_0\{x/t_1\}\ t_2 ... t_n)$$

Iterating the process of head reduction until the head of the reduced term is no more an abstraction leads to the *β-head normal form* of t:

$$t \triangleright ... \triangleright \lambda x_1 : T_1. ... \lambda x_k : T_k.\ (v\ u_1 ... u_m)$$

where v is not an abstraction (nor an application). Note that the head normal form must not be confused with the normal form since some u_i can be reducible. Similar notions of head-normal forms involving δ, ι and ζ reductions or any combination of those can also be defined.

4.4.5 Inductive Definitions

Formally, we can represent any *inductive definition* as $\mathsf{Ind}\ [p]\ (\Gamma_I\ :=\ \Gamma_C)$ where:

- Γ_I determines the names and types of inductive types;
- Γ_C determines the names and types of constructors of these inductive types;
- p determines the number of parameters of these inductive types.

These inductive definitions, together with global assumptions and global definitions, then form the global environment. Additionally, for any p there always exists $\Gamma_P = [a_1 : A_1;\ ...;\ a_p : A_p]$ such that each T in $(t : T) \in \Gamma_I \cup \Gamma_C$ can be written as: $\forall \Gamma_P, T'$ where Γ_P is called the *context of parameters*. Furthermore, we must have that each T in $(t : T) \in \Gamma_I$ can be written as: $\forall \Gamma_P, \forall \Gamma_{Arr(t)}, S$ where $\Gamma_{Arr(t)}$ is called the *Arity* of the inductive type t and S is called the sort of the inductive type t (not to be confused with $\mathcal{S}$ which is the set of sorts).

Example

The declaration for parameterized lists is:

$$\mathsf{Ind}\ [1]\left([\mathsf{list} : \mathsf{Set} \to \mathsf{Set}]\ :=\ \begin{bmatrix} \mathsf{nil} & : & \forall A : \mathsf{Set},\ \mathsf{list}\ A \\ \mathsf{cons} & : & \forall A : \mathsf{Set},\ A \to \mathsf{list}\ A \to \mathsf{list}\ A \end{bmatrix}\right)$$

which corresponds to the result of the Coq declaration:

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

Example

The declaration for a mutual inductive definition of tree and forest is:

$$\mathsf{Ind}\ [0]\left(\begin{bmatrix} \mathsf{tree} & : & \mathsf{Set} \\ \mathsf{forest} & : & \mathsf{Set} \end{bmatrix}\ :=\ \begin{bmatrix} \mathsf{node} & : & \mathsf{forest} \to \mathsf{tree} \\ \mathsf{emptyf} & : & \mathsf{forest} \\ \mathsf{consf} & : & \mathsf{tree} \to \mathsf{forest} \to \mathsf{forest} \end{bmatrix}\right)$$

which corresponds to the result of the Coq declaration:

```
Inductive tree : Set :=
| node : forest -> tree
with forest : Set :=
| emptyf : forest
| consf : tree -> forest -> forest.
```

Example

The declaration for a mutual inductive definition of even and odd is:

$$\mathsf{Ind}\ [0]\left(\begin{bmatrix} \mathsf{even} & : & \mathsf{nat} \to \mathsf{Prop} \\ \mathsf{odd} & : & \mathsf{nat} \to \mathsf{Prop} \end{bmatrix}\ :=\ \begin{bmatrix} \mathsf{even}_{\mathsf{O}} & : & \mathsf{even}\ 0 \\ \mathsf{even}_{\mathsf{S}} & : & \forall n,\ \mathsf{odd}\ n \to \mathsf{even}\ (\mathsf{S}\ n) \\ \mathsf{odd}_{\mathsf{S}} & : & \forall n,\ \mathsf{even}\ n \to \mathsf{odd}\ (\mathsf{S}\ n) \end{bmatrix}\right)$$

which corresponds to the result of the Coq declaration:

```
Inductive even : nat -> Prop :=
| even_O : even 0
| even_S : forall n, odd n -> even (S n)
with odd : nat -> Prop :=
| odd_S : forall n, even n -> odd (S n).
```

Types of inductive objects

We have to give the type of constants in a global environment E which contains an inductive definition.

Ind

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C) \in E \qquad (a : A) \in \Gamma_I}{E[\Gamma] \vdash a : A}$$

Constr

$$\frac{\mathcal{WF}(E)[\Gamma] \qquad \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C) \in E \qquad (c : C) \in \Gamma_C}{E[\Gamma] \vdash c : C}$$

Example

Provided that our environment E contains inductive definitions we showed before, these two inference rules above enable us to conclude that:

$$E[\Gamma] \vdash \mathsf{even} : \mathsf{nat} \to \mathsf{Prop}$$
$$E[\Gamma] \vdash \mathsf{odd} : \mathsf{nat} \to \mathsf{Prop}$$
$$E[\Gamma] \vdash \mathsf{even}_\mathsf{O} : \mathsf{even}\ \mathsf{O}$$
$$E[\Gamma] \vdash \mathsf{even}_\mathsf{S} : \forall n : \mathsf{nat},\ \mathsf{odd}\ n \to \mathsf{even}\ (\mathsf{S}\ n)$$
$$E[\Gamma] \vdash \mathsf{odd}_\mathsf{S} : \forall n : \mathsf{nat},\ \mathsf{even}\ n \to \mathsf{odd}\ (\mathsf{S}\ n)$$

Well-formed inductive definitions

We cannot accept any inductive definition because some of them lead to inconsistent systems. We restrict ourselves to definitions which satisfy a syntactic criterion of positivity. Before giving the formal rules, we need a few definitions:

Arity of a given sort

A type T is an *arity of sort* s if it converts to the sort s or to a product $\forall x : T,\ U$ with U an arity of sort s.

Example

$A \to \mathsf{Set}$ is an arity of sort Set. $\forall A : \mathsf{Prop},\ A \to \mathsf{Prop}$ is an arity of sort Prop.

Arity

A type T is an *arity* if there is a $s \in \mathcal{S}$ such that T is an arity of sort s.

Example

$A \to \mathsf{Set}$ and $\forall A : \mathsf{Prop},\ A \to \mathsf{Prop}$ are arities.

Type of constructor

We say that T is a *type of constructor of* I in one of the following two cases:

- T is $(I\ t_1 ... t_n)$
- T is $\forall x : U,\ T'$ where T' is also a type of constructor of I

Example

nat and $\mathsf{nat} \to \mathsf{nat}$ are types of constructor of nat. $\forall A : \mathsf{Type},\ \mathsf{list}\ A$ and $\forall A : \mathsf{Type},\ A \to \mathsf{list}\ A \to \mathsf{list}\ A$ are types of constructor of list.

Positivity Condition

The type of constructor T will be said to *satisfy the positivity condition* for a constant X in the following cases:

- $T = (X\ t_1 ... t_n)$ and X does not occur free in any t_i
- $T = \forall x : U,\ V$ and X occurs only strictly positively in U and the type V satisfies the positivity condition for X.

Strict positivity

The constant X *occurs strictly positively* in T in the following cases:

- X does not occur in T
- T converts to $(X\ t_1 ... t_n)$ and X does not occur in any of t_i
- T converts to $\forall x : U,\ V$ and X does not occur in type U but occurs strictly positively in type V
- T converts to $(I\ a_1 ... a_m\ t_1 ... t_p)$ where I is the name of an inductive definition of the form

$$\mathsf{Ind}\ [m]\,(I : A\ :=\ c_1 : \forall p_1 : P_1, ... \forall p_m : P_m,\ C_1;\ ...;\ c_n : \forall p_1 : P_1, ... \forall p_m : P_m,\ C_n)$$

  (in particular, it is not mutually defined and it has m parameters) and X does not occur in any of the t_i, and the (instantiated) types of constructor $C_i\{p_j/a_j\}_{j=1...m}$ of I satisfy the nested positivity condition for X

Nested Positivity

The type of constructor T of I *satisfies the nested positivity condition* for a constant X in the following cases:

- $T = (I\ b_1 ... b_m\ u_1 ... u_p)$, I is an inductive type with m parameters and X does not occur in any u_i
- $T = \forall x : U,\ V$ and X occurs only strictly positively in U and the type V satisfies the nested positivity condition for X

Example

For instance, if one considers the following variant of a tree type branching over the natural numbers:

```
Inductive nattree (A:Type) : Type :=
| leaf : nattree A
| natnode : A -> (nat -> nattree A) -> nattree A.
```

Then every instantiated constructor of `nattree A` satisfies the nested positivity condition for `nattree`:

- Type `nattree A` of constructor `leaf` satisfies the positivity condition for `nattree` because `nattree` does not appear in any (real) arguments of the type of that constructor (primarily because `nattree` does not have any (real) arguments) ... (bullet 1)
- Type `A` $\to$ `(nat` $\to$ `nattree A)` $\to$ `nattree A` of constructor `natnode` satisfies the positivity condition for `nattree` because:
  - `nattree` occurs only strictly positively in `A` ... (bullet 1)
  - `nattree` occurs only strictly positively in `nat` $\to$ `nattree A` ... (bullet 3 + 2)
  - `nattree` satisfies the positivity condition for `nattree A` ... (bullet 1)

Correctness rules

We shall now describe the rules allowing the introduction of a new inductive definition.

Let E be a global environment and Γ_P, Γ_I, Γ_C be contexts such that Γ_I is $[I_1 : \forall \Gamma_P, A_1;\ ...;\ I_k : \forall \Gamma_P, A_k]$, and Γ_C is $[c_1 : \forall \Gamma_P, C_1;\ ...;\ c_n : \forall \Gamma_P, C_n]$. Then

W-Ind

$$\frac{\mathcal{WF}(E)[\Gamma_P] \qquad (E[\Gamma_I; \Gamma_P] \vdash C_i : s_{q_i})_{i=1...n}}{\mathcal{WF}(E;\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C))[]}$$

provided that the following side conditions hold:

- $k > 0$ and all of I_j and c_i are distinct names for $j = 1...k$ and $i = 1...n$,
- p is the number of parameters of $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$ and Γ_P is the context of parameters,
- for $j = 1...k$ we have that A_j is an arity of sort s_j and $I_j \notin E$,
- for $i = 1...n$ we have that C_i is a type of constructor of I_{q_i} which satisfies the positivity condition for $I_1 ... I_k$ and $c_i \notin E$.

One can remark that there is a constraint between the sort of the arity of the inductive type and the sort of the type of its constructors which will always be satisfied for the impredicative sorts SProp and Prop but may fail to define inductive type on sort Set and generate constraints between universes for inductive types in the Type hierarchy.

Example

It is well known that the existential quantifier can be encoded as an inductive definition. The following declaration introduces the second-order existential quantifier $\exists X.P(X)$.

```
Inductive exProp (P:Prop->Prop) : Prop :=
| exP_intro : forall X:Prop, P X -> exProp P.
```

The same definition on Set is not allowed and fails:

```
Fail Inductive exSet (P:Set->Prop) : Set :=
exS_intro : forall X:Set, P X -> exSet P.
    The command has indeed failed with message:
    Large non-propositional inductive types must be in Type.
```

It is possible to declare the same inductive definition in the universe Type. The exType inductive definition has type $(\mathsf{Type}(i) \to \mathsf{Prop}) \to \mathsf{Type}(j)$ with the constraint that the parameter X of $\mathsf{exT}_{\mathsf{intro}}$ has type $\mathsf{Type}(k)$ with $k < j$ and $k \leq i$.

```
Inductive exType (P:Type->Prop) : Type :=
exT_intro : forall X:Type, P X -> exType P.
    exType is defined
    exType_rect is defined
    exType_ind is defined
    exType_rec is defined
    exType_sind is defined
```

Example: Negative occurrence (first example)

The following inductive definition is rejected because it does not satisfy the positivity condition:

```
Fail Inductive I : Prop := not_I_I (not_I : I -> False) : I.
    The command has indeed failed with message:
    Non strictly positive occurrence of "I" in "(I -> False) -> I".
```

If we were to accept such definition, we could derive a contradiction from it (we can test this by disabling the *Positivity Checking* flag):

```
Definition I_not_I : I -> ~ I := fun i =>
  match i with not_I_I not_I => not_I end.
    I_not_I is defined

Lemma contradiction : False.
Proof.
enough (I /\ ~ I) as [] by contradiction.
split.
- apply not_I_I.
intro.
now apply I_not_I.
- intro.
now apply I_not_I.
Qed.
```

Example: Negative occurrence (second example)

Here is another example of an inductive definition which is rejected because it does not satify the positivity condition:

```
Fail Inductive Lam := lam (_ : Lam -> Lam).
    The command has indeed failed with message:
    Non strictly positive occurrence of "Lam" in "(Lam -> Lam) -> Lam".
```

Again, if we were to accept it, we could derive a contradiction (this time through a non-terminating recursive function):

```
Fixpoint infinite_loop l : False :=
  match l with lam x => infinite_loop (x l) end.
    infinite_loop is defined
    infinite_loop is recursively defined (decreasing on 1st argument)

Check infinite_loop (lam (@id Lam)) : False.
    infinite_loop (lam (id (A:=Lam))) : False
         : False
```

Template polymorphism

Inductive types can be made polymorphic over the universes introduced by their parameters in Type, if the minimal inferred sort of the inductive declarations either mention some of those parameter universes or is computed to be Prop or Set.

If A is an arity of some sort and s is a sort, we write $A_{/s}$ for the arity obtained from A by replacing its sort with s. Especially, if A is well-typed in some global environment and local context, then $A_{/s}$ is typable by typability of all products in the Calculus of Inductive Constructions. The following typing rule is added to the theory.

Let $\mathsf{Ind}\,[p]\,(\Gamma_I\ :=\ \Gamma_C)$ be an inductive definition. Let $\Gamma_P = [p_1 : P_1;\ ...;\ p_p : P_p]$ be its context of parameters, $\Gamma_I = [I_1 : \forall \Gamma_P, A_1;\ ...;\ I_k : \forall \Gamma_P, A_k]$ its context of definitions and $\Gamma_C = [c_1 : \forall \Gamma_P, C_1;\ ...;\ c_n : \forall \Gamma_P, C_n]$ its context of constructors, with c_i a constructor of I_{q_i}. Let $m \leq p$ be the length of the longest prefix of parameters such that the m first arguments of all occurrences of all I_j in all C_k (even the occurrences in the hypotheses of C_k) are exactly applied to $p_1 ... p_m$ (m is the number of *recursively uniform parameters* and the $p - m$ remaining parameters are the *recursively non-uniform parameters*). Let $q_1, ..., q_r$, with $0 \leq r \leq m$, be a (possibly) partial instantiation of the recursively uniform parameters of Γ_P. We have:

Ind-Family

$$\frac{\left\{\begin{array}{l}\mathsf{Ind}\,[p]\,(\Gamma_I\ :=\ \Gamma_C) \in E\\ (E[] \vdash q_l : P'_l)_{l=1...r}\\ (E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1...l-1})_{l=1...r}\\ 1 \leq j \leq k\end{array}\right.}{E[] \vdash I_j\ q_1 ... q_r : \forall [p_{r+1} : P_{r+1};\ ...;\ p_p : P_p], (A_j)_{/s_j}}$$

provided that the following side conditions hold:

- $\Gamma_{P'}$ is the context obtained from Γ_P by replacing each P_l that is an arity with P'_l for $1 \leq l \leq r$ (notice that P_l arity implies P'_l arity since $E[] \vdash P'_l \leq_{\beta\delta\iota\zeta\eta} P_l\{p_u/q_u\}_{u=1...l-1}$);
- there are sorts s_i, for $1 \leq i \leq k$ such that, for $\Gamma_{I'} = [I_1 : \forall \Gamma_{P'}, (A_1)_{/s_1};\ ...;\ I_k : \forall \Gamma_{P'}, (A_k)_{/s_k}]$ we have $(E[\Gamma_{I'}; \Gamma_{P'}] \vdash C_i : s_{q_i})_{i=1...n}$;

- the sorts s_i are all introduced by the inductive declaration and have no universe constraints beside being greater than or equal to Prop, and such that all eliminations, to Prop, Set and Type(j), are allowed (see Section *Destructors*).

Notice that if $I_j\ q_1...q_r$ is typable using the rules **Ind-Const** and **App**, then it is typable using the rule **Ind-Family**. Conversely, the extended theory is not stronger than the theory without **Ind-Family**. We get an equiconsistency result by mapping each $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$ occurring into a given derivation into as many different inductive types and constructors as the number of different (partial) replacements of sorts, needed for this derivation, in the parameters that are arities (this is possible because $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$ well-formed implies that $\mathsf{Ind}\ [p]\,(\Gamma_{I'}\ :=\ \Gamma_{C'})$ is well-formed and has the same allowed eliminations, where $\Gamma_{I'}$ is defined as above and $\Gamma_{C'} = [c_1 : \forall\Gamma_{P'}, C_1;\ ...;\ c_n : \forall\Gamma_{P'}, C_n])$. That is, the changes in the types of each partial instance $q_1...q_r$ can be characterized by the ordered sets of arity sorts among the types of parameters, and to each signature is associated a new inductive definition with fresh names. Conversion is preserved as any (partial) instance $I_j\ q_1...q_r$ or $C_i\ q_1...q_r$ is mapped to the names chosen in the specific instance of $\mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C)$.

Warning: The restriction that sorts are introduced by the inductive declaration prevents inductive types declared in sections to be template-polymorphic on universes introduced previously in the section: they cannot parameterize over the universes introduced with section variables that become parameters at section closing time, as these may be shared with other definitions from the same section which can impose constraints on them.

Flag: `Auto Template Polymorphism`
This flag, enabled by default, makes every inductive type declared at level Type (without annotations or hiding it behind a definition) template polymorphic if possible.

This can be prevented using the `universes(notemplate)` attribute.

Warning: `Automatically declaring` *ident* `as template polymorphic.`
Warning `auto-template` can be used to find which types are implicitly declared template polymorphic by *Auto Template Polymorphism*.

An inductive type can be forced to be template polymorphic using the `universes(template)` attribute: it should then fulfill the criterion to be template polymorphic or an error is raised.

Error: `Inductive` *ident* `cannot be made template polymorphic.`
This error is raised when the `#[universes(template)]` attribute is on but the inductive cannot be made polymorphic on any universe or be inferred to live in Prop or Set.

Template polymorphism and universe polymorphism (see Chapter *Polymorphic Universes*) are incompatible, so if the later is enabled it will prevail over automatic template polymorphism and cause an error when using the `universes(template)` attribute.

Flag: `Template Check`
This flag is on by default. Turning it off disables the check of locality of the sorts when abstracting the inductive over its parameters. This is a deprecated and *unsafe* flag that can introduce inconsistencies, it is only meant to help users incrementally update code from Coq versions < 8.10 which did not implement this check. The `Coq89.v` compatibility file sets this flag globally. A global `-no-template-check` command line option is also available. Use at your own risk. Use of this flag is recorded in the typing flags associated to a definition but is *not* supported by the Coq checker (`coqchk`). It will appear in `Print Assumptions` and `About @ident` output involving inductive declarations that were (potentially unsoundly) assumed to be template polymorphic.

In practice, the rule **Ind-Family** is used by Coq only when all the inductive types of the inductive definition are declared with an arity whose sort is in the Type hierarchy. Then, the polymorphism is over the parameters whose type is an arity of sort in the Type hierarchy. The sorts s_j are chosen canonically so that each s_j is minimal with respect to the hierarchy $\mathsf{Prop} \subset \mathsf{Set}_p \subset \mathsf{Type}$ where Set_p is predicative Set. More precisely, an

empty or small singleton inductive definition (i.e. an inductive definition of which all inductive types are singleton – see Section *Destructors*) is set in Prop, a small non-singleton inductive type is set in Set (even in case Set is impredicative – see Section *The-Calculus-of-Inductive-Construction-with-impredicative-Set*), and otherwise in the Type hierarchy.

Note that the side-condition about allowed elimination sorts in the rule **Ind-Family** avoids to recompute the allowed elimination sorts at each instance of a pattern matching (see Section *Destructors*). As an example, let us consider the following definition:

Example

```
Inductive option (A:Type) : Type :=
| None : option A
| Some : A -> option A.
```

As the definition is set in the Type hierarchy, it is used polymorphically over its parameters whose types are arities of a sort in the Type hierarchy. Here, the parameter A has this property, hence, if `option` is applied to a type in Set, the result is in Set. Note that if `option` is applied to a type in Prop, then, the result is not set in Prop but in Set still. This is because `option` is not a singleton type (see Section *Destructors*) and it would lose the elimination to Set and Type if set in Prop.

Example

```
Check (fun A:Set => option A).
    fun A : Set => option A
         : Set -> Set

Check (fun A:Prop => option A).
    fun A : Prop => option A
         : Prop -> Set
```

Here is another example.

Example

```
Inductive prod (A B:Type) : Type := pair : A -> B -> prod A B.
```

As `prod` is a singleton type, it will be in Prop if applied twice to propositions, in Set if applied twice to at least one type in Set and none in Type, and in Type otherwise. In all cases, the three kind of eliminations schemes are allowed.

Example

```
Check (fun A:Set => prod A).
    fun A : Set => prod A
         : Set -> Type -> Type

Check (fun A:Prop => prod A A).
    fun A : Prop => prod A A
         : Prop -> Prop
```

(continues on next page)

(continued from previous page)

```
Check (fun (A:Prop) (B:Set) => prod A B).
    fun (A : Prop) (B : Set) => prod A B
         : Prop -> Set -> Set

Check (fun (A:Type) (B:Prop) => prod A B).
    fun (A : Type) (B : Prop) => prod A B
         : Type -> Prop -> Type
```

Note: Template polymorphism used to be called "sort-polymorphism of inductive types" before universe polymorphism (see Chapter *Polymorphic Universes*) was introduced.

Destructors

The specification of inductive definitions with arities and constructors is quite natural. But we still have to say how to use an object in an inductive type.

This problem is rather delicate. There are actually several different ways to do that. Some of them are logically equivalent but not always equivalent from the computational point of view or from the user point of view.

From the computational point of view, we want to be able to define a function whose domain is an inductively defined type by using a combination of case analysis over the possible constructors of the object and recursion.

Because we need to keep a consistent theory and also we prefer to keep a strongly normalizing reduction, we cannot accept any sort of recursion (even terminating). So the basic idea is to restrict ourselves to primitive recursive functions and functionals.

For instance, assuming a parameter $A : \mathsf{Set}$ exists in the local context, we want to build a function length of type $\mathsf{list}\ A \to \mathsf{nat}$ which computes the length of the list, such that $(\mathsf{length}\ (\mathsf{nil}\ A)) = \mathsf{O}$ and $(\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)) = (\mathsf{S}\ (\mathsf{length}\ l))$. We want these equalities to be recognized implicitly and taken into account in the conversion rule.

From the logical point of view, we have built a type family by giving a set of constructors. We want to capture the fact that we do not have any other way to build an object in this type. So when trying to prove a property about an object m in an inductive type it is enough to enumerate all the cases where m starts with a different constructor.

In case the inductive definition is effectively a recursive one, we want to capture the extra property that we have built the smallest fixed point of this recursive equation. This says that we are only manipulating finite objects. This analysis provides induction principles. For instance, in order to prove $\forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l))$ it is enough to prove:

- $(\mathsf{has\_length}\ A\ (\mathsf{nil}\ A)\ (\mathsf{length}\ (\mathsf{nil}\ A)))$
- $\forall a : A,\ \forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l)) \to (\mathsf{has\_length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{length}\ (\mathsf{cons}\ A\ a\ l)))$

which given the conversion equalities satisfied by length is the same as proving:

- $(\mathsf{has\_length}\ A\ (\mathsf{nil}\ A)\ \mathsf{O})$
- $\forall a : A,\ \forall l : \mathsf{list}\ A,\ (\mathsf{has\_length}\ A\ l\ (\mathsf{length}\ l)) \to (\mathsf{has\_length}\ A\ (\mathsf{cons}\ A\ a\ l)\ (\mathsf{S}\ (\mathsf{length}\ l)))$

One conceptually simple way to do that, following the basic scheme proposed by Martin-Löf in his Intuitionistic Type Theory, is to introduce for each inductive definition an elimination operator. At the logical level it is a proof of the usual induction principle and at the computational level it implements a generic operator for doing primitive recursion over the structure.

But this operator is rather tedious to implement and use. We choose in this version of Coq to factorize the operator for primitive recursion into two more primitive operations as was first suggested by Th. Coquand in *[Coq92]*. One is the definition by pattern matching. The second one is a definition by guarded fixpoints.

The match ... with ... end construction

The basic idea of this operator is that we have an object m in an inductive type I and we want to prove a property which possibly depends on m. For this, it is enough to prove the property for $m = (c_i\ u_1 ... u_{p_i})$ for each constructor of I. The Coq term for this proof will be written:

$$\mathsf{match}\ m\ \mathsf{with}\ (c_1\ x_{11} ... x_{1p_1}) \Rightarrow f_1 | ... | (c_n\ x_{n1} ... x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

In this expression, if m eventually happens to evaluate to $(c_i\ u_1 ... u_{p_i})$ then the expression will behave as specified in its i-th branch and it will reduce to f_i where the $x_{i1} ... x_{ip_i}$ are replaced by the $u_1 ... u_{p_i}$ according to the ι-reduction.

Actually, for type checking a match...with...end expression we also need to know the predicate P to be proved by case analysis. In the general case where I is an inductively defined n-ary relation, P is a predicate over $n+1$ arguments: the n first ones correspond to the arguments of I (parameters excluded), and the last one corresponds to object m. Coq can sometimes infer this predicate but sometimes not. The concrete syntax for describing this predicate uses the as...in...return construction. For instance, let us assume that I is an unary predicate with one parameter and one argument. The predicate is made explicit using the syntax:

$$\mathsf{match}\ m\ \mathsf{as}\ x\ \mathsf{in}\ I\ \_\ a\ \mathsf{return}\ P\ \mathsf{with}\ (c_1\ x_{11} ... x_{1p_1}) \Rightarrow f_1 | ... | (c_n\ x_{n1} ... x_{np_n}) \Rightarrow f_n\ \mathsf{end}$$

The as part can be omitted if either the result type does not depend on m (non-dependent elimination) or m is a variable (in this case, m can occur in P where it is considered a bound variable). The in part can be omitted if the result type does not depend on the arguments of I. Note that the arguments of I corresponding to parameters *must* be _, because the result type is not generalized to all possible values of the parameters. The other arguments of I (sometimes called indices in the literature) have to be variables (a above) and these variables can occur in P. The expression after in must be seen as an *inductive type pattern*. Notice that expansion of implicit arguments and notations apply to this pattern. For the purpose of presenting the inference rules, we use a more compact notation:

$$\mathsf{case}(m, (\lambda ax.P), \lambda x_{11} ... x_{1p_1}.f_1\ |...|\ \lambda x_{n1} ... x_{np_n}.f_n)$$

Allowed elimination sorts. An important question for building the typing rule for match is what can be the type of $\lambda ax.P$ with respect to the type of m. If $m : I$ and $I : A$ and $\lambda ax.P : B$ then by $[I : A|B]$ we mean that one can use $\lambda ax.P$ with m in the above match-construct.

Notations. The $[I : A|B]$ is defined as the smallest relation satisfying the following rules: We write $[I|B]$ for $[I : A|B]$ where A is the type of I.

The case of inductive types in sorts Set or Type is simple. There is no restriction on the sort of the predicate to be eliminated.

Prod

$$\frac{[(I\ x) : A'|B']}{[I : \forall x : A,\ A'|\forall x : A,\ B']}$$

Set & Type

$$\frac{s_1 \in \{\mathsf{Set}, \mathsf{Type}(j)\} \qquad s_2 \in \mathcal{S}}{[I : s_1|I \to s_2]}$$

The case of Inductive definitions of sort Prop is a bit more complicated, because of our interpretation of this sort. The only harmless allowed eliminations, are the ones when predicate P is also of sort Prop or is of the morally smaller sort SProp.

Prop

$$\frac{s \in \{\mathsf{SProp}, \mathsf{Prop}\}}{[I : \mathsf{Prop} | I \to s]}$$

Prop is the type of logical propositions, the proofs of properties P in Prop could not be used for computation and are consequently ignored by the extraction mechanism. Assume A and B are two propositions, and the logical disjunction $A \vee B$ is defined inductively by:

Example

```
Inductive or (A B:Prop) : Prop :=
or_introl : A -> or A B | or_intror : B -> or A B.
```

The following definition which computes a boolean value by case over the proof of `or A B` is not accepted:

Example

```
Fail Definition choice (A B: Prop) (x:or A B) :=
match x with or_introl _ _ a => true | or_intror _ _ b => false end.
    The command has indeed failed with message:
    Incorrect elimination of "x" in the inductive type "or":
    the return type has sort "Set" while it should be "SProp" or "Prop".
    Elimination of an inductive object of sort Prop
    is not allowed on a predicate in sort Set
    because proofs can be eliminated only to build proofs.
```

From the computational point of view, the structure of the proof of `(or A B)` in this term is needed for computing the boolean value.

In general, if I has type Prop then P cannot have type $I \to$ Set, because it will mean to build an informative proof of type $(P\ m)$ doing a case analysis over a non-computational object that will disappear in the extracted program. But the other way is safe with respect to our interpretation we can have I a computational object and P a non-computational one, it just corresponds to proving a logical property of a computational object.

In the same spirit, elimination on P of type $I \to$ Type cannot be allowed because it trivially implies the elimination on P of type $I \to$ Set by cumulativity. It also implies that there are two proofs of the same property which are provably different, contradicting the proof-irrelevance property which is sometimes a useful axiom:

Example

```
Axiom proof_irrelevance : forall (P : Prop) (x y : P), x=y.
    proof_irrelevance is declared
```

The elimination of an inductive type of sort Prop on a predicate P of type $I \to$ Type leads to a paradox when applied to impredicative inductive definition like the second-order existential quantifier `exProp` defined above, because it gives access to the two projections on this type.

Empty and singleton elimination. There are special inductive definitions in Prop for which more eliminations are allowed.

Prop-extended

$$\frac{I \text{ is an empty or singleton definition} \qquad s \in \mathcal{S}}{[I : \mathsf{Prop} | I \to s]}$$

A *singleton definition* has only one constructor and all the arguments of this constructor have type Prop. In that case, there is a canonical way to interpret the informative extraction on an object in that type, such that the elimination on any sort s is legal. Typical examples are the conjunction of non-informative propositions and the equality. If there is a hypothesis $h : a = b$ in the local context, it can be used for rewriting not only in logical propositions but also in any type.

Example

```
Print eq_rec.
    eq_rec =
    fun (A : Type) (x : A) (P : A -> Set) => eq_rect x P
         : forall (A : Type) (x : A) (P : A -> Set),
           P x -> forall y : A, x = y -> P y

    Arguments eq_rec [A]%type_scope _ _%function_scope

Require Extraction.
    [Loading ML file extraction_plugin.cmxs ... done]

Extraction eq_rec.
    (** val eq_rec : 'a1 -> 'a2 -> 'a1 -> 'a2 **)

    let eq_rec _ f _ =
      f
```

An empty definition has no constructors, in that case also, elimination on any sort is allowed.

Inductive types in SProp must have no constructors (i.e. be empty) to be eliminated to produce relevant values.

Note that thanks to proof irrelevance elimination functions can be produced for other types, for instance the elimination for a unit type is the identity.

Type of branches. Let c be a term of type C, we assume C is a type of constructor for an inductive type I. Let P be a term that represents the property to be proved. We assume r is the number of parameters and s is the number of arguments.

We define a new type $\{c : C\}^P$ which represents the type of the branch corresponding to the $c : C$ constructor.

$$\begin{array}{ll} \{c : (I\ q_1 \ldots q_r\ t_1 \ldots t_s)\}^P & \equiv (P\ t_1 \ldots\ t_s\ c) \\ \{c : \forall x : T,\ C\}^P & \equiv \forall x : T,\ \{(c\ x) : C\}^P \end{array}$$

We write $\{c\}^P$ for $\{c : C\}^P$ with C the type of c.

Example

The following term in concrete syntax:

```
match t as l return P' with
| nil _ => t1
| cons _ hd tl => t2
end
```

can be represented in abstract syntax as

$$\mathsf{case}(t, P, f_1|f_2)$$

where

$$\begin{array}{rcl} P & = & \lambda l.\ P' \\ f_1 & = & t_1 \\ f_2 & = & \lambda(hd : \mathsf{nat}).\ \lambda(tl : \mathsf{list\ nat}).\ t_2 \end{array}$$

According to the definition:

$$\{(\mathsf{nil\ nat})\}^P \equiv \{(\mathsf{nil\ nat}) : (\mathsf{list\ nat})\}^P \equiv (P\ (\mathsf{nil\ nat}))$$

$$\begin{array}{rl} \{(\mathsf{cons\ nat})\}^P & \equiv \{(\mathsf{cons\ nat}) : (\mathsf{nat} \to \mathsf{list\ nat} \to \mathsf{list\ nat})\}^P \\ & \equiv \forall n : \mathsf{nat},\ \{(\mathsf{cons\ nat}\ n) : (\mathsf{list\ nat} \to \mathsf{list\ nat})\}^P \\ & \equiv \forall n : \mathsf{nat},\ \forall l : \mathsf{list\ nat},\ \{(\mathsf{cons\ nat}\ n\ l) : (\mathsf{list\ nat})\}^P \\ & \equiv \forall n : \mathsf{nat},\ \forall l : \mathsf{list\ nat},\ (P\ (\mathsf{cons\ nat}\ n\ l)). \end{array}$$

Given some P then $\{(\mathsf{nil\ nat})\}^P$ represents the expected type of f_1, and $\{(\mathsf{cons\ nat})\}^P$ represents the expected type of f_2.

Typing rule. Our very general destructor for inductive definition enjoys the following typing rule

match

$$\frac{\begin{array}{l} E[\Gamma] \vdash c : (I\ q_1 ... q_r\ t_1 ... t_s) \\ E[\Gamma] \vdash P : B \\ [(I\ q_1 ... q_r)|B] \\ (E[\Gamma] \vdash f_i : \{(c_{p_i}\ q_1 ... q_r)\}^P)_{i=1...l} \end{array}}{E[\Gamma] \vdash \mathsf{case}(c, P, f_1|...|f_l) : (P\ t_1 ... t_s\ c)}$$

provided I is an inductive type in a definition $\mathsf{Ind}\ [r]\ (\Gamma_I\ :=\ \Gamma_C)$ with $\Gamma_C = [c_1 : C_1;\ ...;\ c_n : C_n]$ and $c_{p_1} ... c_{p_l}$ are the only constructors of I.

Example

Below is a typing rule for the term shown in the previous example:

list example

$$\frac{\begin{array}{l} E[\Gamma] \vdash t : (\mathsf{list\ nat}) \\ E[\Gamma] \vdash P : B \\ [(\mathsf{list\ nat})|B] \\ E[\Gamma] \vdash f_1 : \{(\mathsf{nil\ nat})\}^P \\ E[\Gamma] \vdash f_2 : \{(\mathsf{cons\ nat})\}^P \end{array}}{E[\Gamma] \vdash \mathsf{case}(t, P, f_1|f_2) : (P\ t)}$$

Definition of ι-reduction. We still have to define the ι-reduction in the general case.

An ι-redex is a term of the following form:

$$\mathsf{case}((c_{p_i}\ q_1...q_r\ a_1...a_m), P, f_1|...|f_l)$$

with c_{p_i} the i-th constructor of the inductive type I with r parameters.

The ι-contraction of this term is $(f_i\ a_1...a_m)$ leading to the general reduction rule:

$$\mathsf{case}((c_{p_i}\ q_1...q_r\ a_1...a_m), P, f_1|...|f_l) \triangleright_\iota (f_i\ a_1...a_m)$$

Fixpoint definitions

The second operator for elimination is fixpoint definition. This fixpoint may involve several mutually recursive definitions. The basic concrete syntax for a recursive set of mutually recursive declarations is (with Γ_i contexts):

$$\mathsf{fix}\ f_1(\Gamma_1) : A_1 := t_1\ \mathsf{with ... with}\ f_n(\Gamma_n) : A_n := t_n$$

The terms are obtained by projections from this set of declarations and are written

$$\mathsf{fix}\ f_1(\Gamma_1) : A_1 := t_1\ \mathsf{with ... with}\ f_n(\Gamma_n) : A_n := t_n\ \mathsf{for}\ f_i$$

In the inference rules, we represent such a term by

$$\mathsf{Fix}\ f_i\{f_1 : A'_1 := t'_1 ... f_n : A'_n := t'_n\}$$

with t'_i (resp. A'_i) representing the term t_i abstracted (resp. generalized) with respect to the bindings in the context Γ_i, namely $t'_i = \lambda\Gamma_i.t_i$ and $A'_i = \forall\Gamma_i, A_i$.

Typing rule

The typing rule is the expected one for a fixpoint.

Fix

$$\frac{(E[\Gamma] \vdash A_i : s_i)_{i=1...n} \qquad (E[\Gamma;\ f_1 : A_1;\ ...;\ f_n : A_n] \vdash t_i : A_i)_{i=1...n}}{E[\Gamma] \vdash \mathsf{Fix}\ f_i\{f_1 : A_1 := t_1 ... f_n : A_n := t_n\} : A_i}$$

Any fixpoint definition cannot be accepted because non-normalizing terms allow proofs of absurdity. The basic scheme of recursion that should be allowed is the one needed for defining primitive recursive functionals. In that case the fixpoint enjoys a special syntactic restriction, namely one of the arguments belongs to an inductive type, the function starts with a case analysis and recursive calls are done on variables coming from patterns and representing subterms. For instance in the case of natural numbers, a proof of the induction principle of type

$$\forall P : \mathsf{nat} \to \mathsf{Prop},\ (P\ \mathsf{O}) \to (\forall n : \mathsf{nat},\ (P\ n) \to (P\ (\mathsf{S}\ n))) \to \forall n : \mathsf{nat},\ (P\ n)$$

can be represented by the term:

$$\lambda P : \mathsf{nat} \to \mathsf{Prop}.\ \lambda f : (P\ \mathsf{O}).\ \lambda g : (\forall n : \mathsf{nat},\ (P\ n) \to (P\ (\mathsf{S}\ n))).$$
$$\mathsf{Fix}\ h\{h : \forall n : \mathsf{nat},\ (P\ n) := \lambda n : \mathsf{nat}.\ \mathsf{case}(n, P, f|\lambda p : \mathsf{nat}.\ (g\ p\ (h\ p)))\}$$

Before accepting a fixpoint definition as being correctly typed, we check that the definition is "guarded". A precise analysis of this notion can be found in *[Gimenez94]*. The first stage is to precise on which argument

the fixpoint will be decreasing. The type of this argument should be an inductive type. For doing this, the syntax of fixpoints is extended and becomes

$$\mathsf{Fix}\ f_i\{f_1/k_1 : A_1 := t_1 ... f_n/k_n : A_n := t_n\}$$

where k_i are positive integers. Each k_i represents the index of parameter of f_i, on which f_i is decreasing. Each A_i should be a type (reducible to a term) starting with at least k_i products $\forall y_1 : B_1,\ ...\forall y_{k_i} : B_{k_i},\ A'_i$ and B_{k_i} an inductive type.

Now in the definition t_i, if f_j occurs then it should be applied to at least k_j arguments and the k_j-th argument should be syntactically recognized as structurally smaller than y_{k_i}.

The definition of being structurally smaller is a bit technical. One needs first to define the notion of *recursive arguments of a constructor*. For an inductive definition $\mathsf{Ind}\ [r]\,(\Gamma_I\ :=\ \Gamma_C)$, if the type of a constructor c has the form $\forall p_1 : P_1,\ ...\forall p_r : P_r,\ \forall x_1 : T_1,\ ...\forall x_m : T_m,\ (I_j\ p_1 ... p_r\ t_1 ... t_s)$, then the recursive arguments will correspond to T_i in which one of the I_l occurs.

The main rules for being structurally smaller are the following. Given a variable y of an inductively defined type in a declaration $\mathsf{Ind}\ [r]\,(\Gamma_I\ :=\ \Gamma_C)$ where Γ_I is $[I_1 : A_1;\ ...;\ I_k : A_k]$, and Γ_C is $[c_1 : C_1;\ ...;\ c_n : C_n]$, the terms structurally smaller than y are:

- $(t\ u)$ and $\lambda x : U.\ t$ when t is structurally smaller than y.
- $\mathsf{case}(c, P, f_1 ... f_n)$ when each f_i is structurally smaller than y. If c is y or is structurally smaller than y, its type is an inductive type I_p part of the inductive definition corresponding to y. Each f_i corresponds to a type of constructor $C_q \equiv \forall p_1 : P_1,\ ..., \forall p_r : P_r,\ \forall y_1 : B_1,\ ...\forall y_m : B_m,\ (I_p\ p_1 ... p_r\ t_1 ... t_s)$ and can consequently be written $\lambda y_1 : B'_1.\ ...\lambda y_m : B'_m.\ g_i$. ($B'_i$ is obtained from B_i by substituting parameters for variables) the variables y_j occurring in g_i corresponding to recursive arguments B_i (the ones in which one of the I_l occurs) are structurally smaller than y.

The following definitions are correct, we enter them using the *Fixpoint* command and show the internal representation.

Example

```
Fixpoint plus (n m:nat) {struct n} : nat :=
match n with
| O => m
| S p => S (plus p m)
end.
    plus is defined
    plus is recursively defined (decreasing on 1st argument)

Print plus.
    plus =
    fix plus (n m : nat) {struct n} : nat :=
      match n with
      | O => m
      | S p => S (plus p m)
      end
         : nat -> nat -> nat

    Arguments plus (_ _)%nat_scope

Fixpoint lgth (A:Set) (l:list A) {struct l} : nat :=
match l with
| nil _ => O
| cons _ a l' => S (lgth A l')
```

(continues on next page)

(continued from previous page)

```
end.
    lgth is defined
    lgth is recursively defined (decreasing on 2nd argument)

Print lgth.
    lgth =
    fix lgth (A : Set) (l : list A) {struct l} : nat :=
      match l with
      | nil _ => 0
      | cons _ _ l' => S (lgth A l')
      end
         : forall A : Set, list A -> nat

    Arguments lgth _%type_scope

Fixpoint sizet (t:tree) : nat := let (f) := t in S (sizef f)
with sizef (f:forest) : nat :=
match f with
| emptyf => 0
| consf t f => plus (sizet t) (sizef f)
end.
    sizet is defined
    sizef is defined
    sizet, sizef are recursively defined (decreasing respectively on 1st,
    1st arguments)

Print sizet.
    sizet =
    fix sizet (t : tree) : nat := let (f) := t in S (sizef f)
    with sizef (f : forest) : nat :=
      match f with
      | emptyf => 0
      | consf t f0 => plus (sizet t) (sizef f0)
      end
    for sizet
         : tree -> nat
```

Reduction rule

Let F be the set of declarations: $f_1/k_1 : A_1 := t_1 ... f_n/k_n : A_n := t_n$. The reduction for fixpoints is:

$$(\mathsf{Fix}\ f_i\{F\}\ a_1 ... a_{k_i})\ \triangleright_\iota\ t_i\{f_k/\mathsf{Fix}\ f_k\{F\}\}_{k=1...n}\ a_1 ... a_{k_i}$$

when a_{k_i} starts with a constructor. This last restriction is needed in order to keep strong normalization and corresponds to the reduction for primitive recursive operators. The following reductions are now possible:

$$\begin{aligned} \mathsf{plus}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O}))\ (\mathsf{S}\ \mathsf{O}) \quad &\triangleright_\iota \quad \mathsf{S}\ (\mathsf{plus}\ (\mathsf{S}\ \mathsf{O})\ (\mathsf{S}\ \mathsf{O})) \\ &\triangleright_\iota \quad \mathsf{S}\ (\mathsf{S}\ (\mathsf{plus}\ \mathsf{O}\ (\mathsf{S}\ \mathsf{O}))) \\ &\triangleright_\iota \quad \mathsf{S}\ (\mathsf{S}\ (\mathsf{S}\ \mathsf{O})) \end{aligned}$$

Mutual induction

The principles of mutual induction can be automatically generated using the Scheme command described in Section *Generation of induction principles with Scheme*.

4.4.6 Admissible rules for global environments

From the original rules of the type system, one can show the admissibility of rules which change the local context of definition of objects in the global environment. We show here the admissible rules that are used in the discharge mechanism at the end of a section.

Abstraction. One can modify a global declaration by generalizing it over a previously assumed constant c. For doing that, we need to modify the reference to the global declaration in the subsequent global environment and local context by explicitly applying this constant to the constant c.

Below, if Γ is a context of the form $[y_1 : A_1;\ ...;\ y_n : A_n]$, we write $\forall x : U,\ \Gamma\{c/x\}$ to mean $[y_1 : \forall x : U,\ A_1\{c/x\};\ ...;\ y_n : \forall x : U,\ A_n\{c/x\}]$ and $E\{|\Gamma|/|\Gamma|c\}$ to mean the parallel substitution $E\{y_1/(y_1\ c)\}...\{y_n/(y_n\ c)\}$.

First abstracting property:

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ c' := t : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c : U;\ E';\ c' := \lambda x : U.\ t\{c/x\} : \forall x : U,\ T\{c/x\};\ E''\{c'/(c'\ c)\})[\Gamma\{c'/(c'\ c)\}]}$$

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ c' : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c : U;\ E';\ c' : \forall x : U,\ T\{c/x\};\ E''\{c'/(c'\ c)\})[\Gamma\{c'/(c'\ c)\}]}$$

$$\frac{\mathcal{WF}(E;\ c : U;\ E';\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C);\ E'')[\Gamma]}{\mathcal{WF}\ \begin{array}{l}(E;\ c : U;\ E';\ \mathsf{Ind}\ [p+1]\,(\forall x : U,\ \Gamma_I\{c/x\}\ :=\ \forall x : U,\ \Gamma_C\{c/x\});\ E''\{|\Gamma_I;\Gamma_C|/|\Gamma_I;\Gamma_C|c\})\\ [\Gamma\{|\Gamma_I;\Gamma_C|/|\Gamma_I;\Gamma_C|c\}]\end{array}}$$

One can similarly modify a global declaration by generalizing it over a previously defined constant c. Below, if Γ is a context of the form $[y_1 : A_1;\ ...;\ y_n : A_n]$, we write $\Gamma\{c/u\}$ to mean $[y_1 : A_1\{c/u\};\ ...;\ y_n : A_n\{c/u\}]$.

Second abstracting property:

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E';\ c' := t : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c := u : U;\ E';\ c' := (\mathsf{let}\ x := u : U\ \mathsf{in}\ t\{c/x\}) : T\{c/u\};\ E'')[\Gamma]}$$

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E';\ c' : T;\ E'')[\Gamma]}{\mathcal{WF}(E;\ c := u : U;\ E';\ c' : T\{c/u\};\ E'')[\Gamma]}$$

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E';\ \mathsf{Ind}\ [p]\,(\Gamma_I\ :=\ \Gamma_C);\ E'')[\Gamma]}{\mathcal{WF}(E;\ c := u : U;\ E';\ \mathsf{Ind}\ [p]\,(\Gamma_I\{c/u\}\ :=\ \Gamma_C\{c/u\});\ E'')[\Gamma]}$$

Pruning the local context. If one abstracts or substitutes constants with the above rules then it may happen that some declared or defined constant does not occur any more in the subsequent global environment and in the local context. One can consequently derive the following property.

First pruning property:

$$\frac{\mathcal{WF}(E;\ c : U;\ E')[\Gamma] \qquad c\ \mathsf{does\ not\ occur\ in}\ E'\ \mathsf{and}\ \Gamma}{\mathcal{WF}(E; E')[\Gamma]}$$

Second pruning property:

$$\frac{\mathcal{WF}(E;\ c := u : U;\ E')[\Gamma] \qquad c\ \mathsf{does\ not\ occur\ in}\ E'\ \mathsf{and}\ \Gamma}{\mathcal{WF}(E; E')[\Gamma]}$$

4.4.7 Co-inductive types

The implementation contains also co-inductive definitions, which are types inhabited by infinite objects. More information on co-inductive definitions can be found in *[Gimenez95][Gimenez98][GimenezCasteran05]*.

4.4.8 The Calculus of Inductive Constructions with impredicative Set

Coq can be used as a type checker for the Calculus of Inductive Constructions with an impredicative sort Set by using the compiler option `-impredicative-set`. For example, using the ordinary `coqtop` command, the following is rejected,

---

Example

```
Fail Definition id: Set := forall X:Set,X->X.
    The command has indeed failed with message:
    The term "forall X : Set, X -> X" has type "Type"
    while it is expected to have type "Set" (universe inconsistency).
```

---

while it will type check, if one uses instead the `coqtop -impredicative-set` option..

The major change in the theory concerns the rule for product formation in the sort Set, which is extended to a domain in any sort:

ProdImp

$$\frac{E[\Gamma] \vdash T : s \qquad s \in \mathcal{S} \qquad E[\Gamma :: (x : T)] \vdash U : \mathsf{Set}}{E[\Gamma] \vdash \forall x : T,\ U : \mathsf{Set}}$$

This extension has consequences on the inductive definitions which are allowed. In the impredicative system, one can build so-called *large inductive definitions* like the example of second-order existential quantifier (`exSet`).

There should be restrictions on the eliminations which can be performed on such definitions. The elimination rules in the impredicative system for sort Set become:

Set1

$$\frac{s \in \{\mathsf{Prop}, \mathsf{Set}\}}{[I : \mathsf{Set} | I \to s]}$$

Set2

$$\frac{I \text{ is a small inductive definition} \qquad s \in \{\mathsf{Type}(i)\}}{[I : \mathsf{Set} | I \to s]}$$

4.5 The Module System

The module system extends the Calculus of Inductive Constructions providing a convenient way to structure large developments as well as a means of massive abstraction.

4.5.1 Modules and module types

Access path. An access path is denoted by p and can be either a module variable X or, if p' is an access path and id an identifier, then $p'.id$ is an access path.

Structure element. A structure element is denoted by e and is either a definition of a constant, an assumption, a definition of an inductive, a definition of a module, an alias of a module or a module type abbreviation.

Structure expression. A structure expression is denoted by S and can be: