Init

Table of Contents

Initial view of some randomly-selected things in the « PROGRAMME » , which sometimes are generating instances : instead of grammatically-describing some *general* class, oneself sensibly-describes *one instance* of the class which is (almost-)generating all the other instances of the class.

1 Functions : functional-hold, delayed-substitution

Oneself lacks to write and solve the question

```
2 + 3
```

In reality, oneself may want / lack to write and solve many such questions. Therefore oneself may hold / bind some name / identifier such as « a » and « p » and write the functional-hold / function ("abstraction")

```
fun p => fun a => a + p
```

and then (delay-)substitute / instantiate / apply this single expression by many input (outer) parameter « p » other than « 3 » and many input (inner) argument « a » other than « 2 », and get output such as this delayed-substitution ("application")

```
(fun p => fun a => a + p) 3 2
```

Does this solves the initial question ? What is the motivation ?

Primo, one motivation may be to memorize things. Therefore oneself may do some outer programming / commanding of the « COQ » computer, and (top / globally) behold / define some name / identifier « f » which memorize / fold / shorten some common expressions such as :

```
Definition f := fun p => fun a => a + p .
```

```
f is defined
```

Secondo, another motivation may be to confirm / check that whatever has been memorized fulfill some prescribed/wanted classification / property / specification / type, by doing logical « COQ » commands such as :

```
Definition f : nat -> nat -> nat := fun p => fun a => a + p .
```

```
f is defined
```

```
Check f.
```

```
f
     : nat -> nat -> nat
```

in contrast to, which is false,

```
Definition f : nat -> nat -> bool := fun p => fun a => a + p .
```

```
Toplevel input, characters 55-60:
> Definition f : nat -> nat -> bool := fun p => fun a => a + p .
>                                                        ^^^^^
Error:
In environment
p : nat
a : nat
The term "a + p" has type "nat" while it is expected to have type "bool".
```

where « nat » is the more primitive class « 0 | 1 | 2 | 3 | ... » and « bool » is the more primitive class « true | false » and « nat -> nat -> nat » is some more composite class which iterates two function class former (inference « .. -> .. » or classifying inference / quantification « forall .. , .. ») such to classify such functions.

In short : whenever assuming « (x : A) » (« x » is some identifier classified by the class « A ») one has « (b : B) » (« b » is some term classified by the classification « B »), … therefore one may (introduce « forall » by) *form* this term and its classification « ((fun x : A => b) : (forall x : A, B)) » , also written « ((fun x : A => b) : (A -> B)) » when the identifier « x » does not occur/mentionned in the textual description of « B ». This process is named functional-holding / functional-binding / discharge.

And in the absence of some prescribed/wanted classification or in the presence of only partial classification, the « COQ » computer will attempt to infer some full classification whenever possible. This inference may fail because the given expression is non classifiable or the prescribed/wanted classification is not contained/subclass of some possible inferred classification.

Tertio, another motivation may be to get the output in some more primitive resolution / value / normal form, by doing computational « COQ » commands such as :

```
Eval compute in f 3 2.
```

```
= 5
: nat
```

In short : whenever « (f : forall x : A, B) » (« f » is some term formed as above) and « (t : A) » (« t » is some term classified by the class « A »), … therefore one may (eliminate « forall » by) *form* this term and its classification « ((f t) : B[t/x]) », where « (B[t/x]) » signifies the substitution of « t » for « x » in the textual description of « B », also written « ((f t) : B) » when the identifier « x » does not occur/mentionned in the textual description of « B ». This process is named delayed-substitution / application.

Finally, other « COQ » commands query the present memory of the « COQ » computer, such as :

```
About f.
```

```
f : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
f is transparent
Expands to: Constant Top.f
```

```
Print f.
```

```
f = fun p : nat => addn^~ p
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

```
Reset f.
```

```
About f.
```

```
f not a defined object.
```

From the angle of view that computers is the "foundations" of mathematics, one may not delay too much on the mathematical "foundations" of the « COQ » computer.

2 Data : class, constructor functions, destructor function

Oneself may want to classify / type some data, for example : classify « true | false » together and name it « bool » (or « bin ») ; classify « 1 | 2 | 3 | ... » together and name it « nat » …

Primo, one shall say alternatives / cases to (recursively) construct / build data in this class. Each alternative is described by some constructor function which always output into this class. Moveover this constructor function may (recursively) take input from this class. The terminology constructor constant is used in the instance that this constructor function does not take any input.

Secondo, one shall say that these given alternate constructions computationally or logically fulfill / support this class, which is that it is sufficient to focus / touch on these (recursively) constructored data when holding this class, which is that any (random) data in the class may be such (recursively) destructed / eliminated / matched / filtered. This is described by one (grammatical) destructor / match function which always input from this class.

2.1 Binary data

The name of the two constructors are « true » and « false » , and the name of the class / type is « bool » ; and this is how to command « COQ » to memorize such names :

```
Inductive bool := true : bool | false : bool.
```

```
bool is defined
bool_rect is defined
bool_ind is defined
bool_rec is defined
```

And « COQ » defines and memorize additional names « bool_rect » , « bool_ind » , « bool_rec » which are easier decoration shortening of the more primitive same (grammatical) destructor / match function :

```
Print bool_rect.
```

```
bool_rect =
fun (P : bool -> Type) (f : P true) (f0 : P false) (b : bool) =>
if b as b0 return (P b0) then f else f0
     : forall P : bool -> Type, P true -> P false -> forall b : bool, P b

Argument scopes are [function_scope _ _ bool_scope]
```

which says as expected that these given « true | false » alternate constructions computationally or logically fulfill / support this class « bool » … Some instance of this same « match » destruction / filtering / elimination function, using shorter grammar, is :

```
Check (if true then 3 else 2) .
```

```
if true then 3 else 2
     : nat
```

```
Eval compute in ((fun b : bool => (if b then 3 else 2)) false) .
```

```
= 2
: nat
```

The « PROGRAMME » contains some collection of binary / boolean operations that mirror reasoning steps on truth values. The functions are named « negb » (negation) , « orb » (orelse) , « andb » (andthen), « implyb » (branch), correspondingly with notations « ~~ » , « || » , « && » , and « ==> » . The first operator is prefix (non left-recursive parsing) parsed as in « ~~ b » , the last three operators are no-prefix (infix) parsed as in « b1 && b2 » .

For instance, the function « andb » ouputs true the-same-as [the first input is true andthen the second input is true] :

```
Definition andb b1 b2 := if b1 then b2 else false.
```

2.2 Numbers data

Any number is zero or the successor of an existing number :

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

```
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

This command says that the only ways to produce numbers are by using the constant symbol / sign / token « O » (capital « o » letter, to represent « 0 »), or by applying the function symbol « S » to some already existing number. In other words, « O » is some number, « (S O) » is some number, « (S (S O)) » , and so on, and these are the only numbers.

And « COQ » defines and memorize additional names « nat_rect » , « nat_ind » , « nat_rec » which are easier decoration shortening of the more primitive same (grammatical) destructor/match function :

```
Print nat_rect.
```

```
nat_rect =
fun (P : nat -> Type) (f : P 0) (f0 : forall n : nat, P n -> P (succn n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | succn n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (succn n)) -> forall n : nat, P n

Argument scopes are [function_scope _ function_scope nat_scope]
```

which says as expected that these given « O | S (n : nat) » alternate constructions computationally or logically fulfill / support this class « nat » … One shall clarify why this « fix » keytext for « nat » (instead of the ealier only « fun » keytext for « bool ») later.

When interacting with « COQ », oneself will often see decimal notations (« 0 » « 1 » « 2 » « 3 » « 4 » …), but these are only some parsing and printing / display facility provided to the programmer for readability. In other words « O » is printed « 0 » , « (S O) » is printed « 1 » , « (S (S O)) » is printed « 2 » … Programmers may also write decimal numbers to describe values, but these are automatically parsed into terms built with « O » and « S ».

Also, the postfix (infix) « x.+1 » notation is translated as the prefix expression « S x » . The « .+1 » notation binds more strongly (at level 2) than function application (at level 10). Attempt :

```
Print Grammar constr.
```

```
Check fun x => (fun n : nat => n) x.+1 .
Locate ".+1" .
```

```
fun x : nat => id (succn x)
     : nat -> nat

Notation            Scope
"n .+1" := S n       : nat_scope
                      (default interpretation)
```

When computing functions over number input data or deducting lemmas over number subject data, oneself may proceed by touching only the alternative cases form of the data, and therefore by the minimality / inductive / elimination for « nat » , the function or deduction will be indeed over all numbers. For example here is the definition of « beheading « S » » (predecessor) for numbers :

```
Definition pred n :=
  match n with
    O => O
  | S t => t
  end.
```

The branch « O => O » says that when « n » has the alternative zero form « O » then the whole expression is transformed to « O » ; here the left « O » is some filter / pattern and the computation decides whether « n » matches this pattern. The branch « S t => t » says that when « n » has the alternative successor form « S t » then « t » is instantiated/bound by this subterm (tail) of « n » and the whole expression is transformed to « t » ; here « S t » is some filter / pattern containing one named filter-identifier / filter-variable « t » and the computation decides whether « n » matches this pattern.

Each constructor must be covered by some branch and by at most one branch. For example, this attempt is not memorized by the « COQ » computer :

```
Fail Definition wrong (n : nat) :=
match n with 0 => true end.
```

```
The command has indeed failed with message:
Non exhaustive pattern-matching: no clause found for pattern
succn _
```

Memo that the « COQ » (outer) parser and printer (« CAMLP5 ») may prevent programmer fatigue and translate the grammar

```
Definition sameOn_bool_nat b n :=
  match b, n with
    | true, S _ => true
    | _, _ => false
  end.
```

as the same as

```
Reset sameOn_bool_nat.
Definition sameOn_bool_nat b n :=
  match b with
    | true => if n is S t then true else false
    | _ => false
  end.
```

Now some clarification for the « fix » keytext. When computing functions over number input data or deducting lemmas over number subject data, oneself may transform the input arguments to the function or lemma into some other input arguments for the *same* function or lemma. The keytext « fix » says that the same function name may be mentioned. For example here is the definition of concatenation (addition) for numbers :

```
Check
  fix add n m :=
    match n with
      S t => add t (S m)
      | O => m
    end .
```

```
fix add (n m : nat) {struct n} : nat :=
  match n with
  | 0 => m
  | succn t => add t (succn m)
  end
     : nat -> nat -> nat
```

Memo that one of the inner memories / accumulators « n » and « m » shall be degrading / decreasing / structural / terminating, here it is the accumulator « n » which is decreasing (structural, « {struct n} »). And the « COQ » computer is very good at detecting when something is degrading, otherwise there are many other techniques to solves this question of termination …

Now instead of using two (inner) accumulators « n » and « m » , oneself may want to use :

- only one inner accumulator « n » , together with
- pending (outer) effects / computations (the « S » surrounding « (add t) ») instead of changing some second inner memory, together with
- one outer parameter « p » which is *not changed* as memory during computation.

```
Check
  fun p => fix add n :=
    match n with
      S t => S (add t)
      | O => p
    end .
```

```
fun p : nat =>
fix add (n : nat) : nat :=
  match n with
  | 0 => p
  | succn t => succn (add t)
  end
     : nat -> nat -> nat
```

Finally oneself may command « COQ » to memorize this expression :

```
Definition add :=
  fun p => fix add n :=
    match n with
      S t => S (add t)
      | O => p
    end .
```

```
add is defined
```

To prevent programmer fatigue, « COQ » has some alias command « Fixpoint » which does *almost the same* thing :

```
Fixpoint add p n {struct n} :=
    match n with
      S t => S (add p t)
      | O => p
    end .
Print add.
```

```
add is defined
add is recursively defined (decreasing on 2nd argument)

add =
fix add (p n : nat) {struct n} : nat :=
  match n with
  | 0 => p
  | succn t => succn (add p t)
  end
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

or « COQ » has some combination of the « Section » command with the « Fixpoint » command which does *precisely the same* thing, because the « Section » process is for holding / binding (outer) parameters / variables :

```
Section Section1.
  Variable p : nat.
  Fixpoint add n :=
      match n with
        S t => S (add t)
        | O => p
      end .
End Section1.
Print add.
```

```
p is declared

add is defined
add is recursively defined (decreasing on 1st argument)



add =
fun p : nat =>
fix add (n : nat) : nat :=
  match n with
  | 0 => p
  | succn t => succn (add t)
  end
     : nat -> nat -> nat

Argument scopes are [nat_scope nat_scope]
```

```
Reset Section1.
```

Little reminder : the « PROGRAMME » defines instead some function « addn » such that « addn n p » is « add p n » (this latest « add » of « Section1 ») … Moreover the « PROGRAMME » defines addition (named « addn » , infix notation « + »), predecessor (« predn » , postfix notation « .-1 »), doubling (« doublen » , postfix notation « .*2 »), multiplication (« muln » , infix notation « * »), subtraction (« subn » , infix notation « - »), division (« divn » , infix notation « %/ »), modulo (« modn » , infix notation « %% »), exponentiation (« expn » , infix notation « ^ »), equality comparison (« eqn » , infix notation « == »), and order comparison (« leq » , infix notation « <= ») on (natural) numbers.

Many of these functions may be defined by reusing the general function « iter », which is the iterator over any number in « nat » , or by reusing the more-general function « foldr » , which is the iterator over any list in « seq » :

```
Print iter.
Definition add (p : nat) : nat -> nat :=
  fun n : nat =>
    iter n (fun acc : nat => S (acc)) p .
Print foldr.
```

```
iter =
fun (T : Type) (n : nat) (f : T -> T) (x : T) =>
let
```

```
  fix loop (m : nat) : T := match m with
                            | 0 => x
                            | succn i => f (loop i)
                            end in
loop n
     : forall T : Type, nat -> (T -> T) -> T -> T

Argument T is implicit
Argument scopes are [type_scope nat_scope function_scope _]

add is defined

foldr =
fun (T R : Type) (f : T -> R -> R) (z0 : R) =>
fix foldr (s : seq T) : R :=
  match s with
  | [::] => z0
  | x :: s' => f x (foldr s')
  end
     : forall T R : Type, (T -> R -> R) -> R -> seq T -> R

Arguments T, R are implicit and maximally inserted
Argument scopes are [type_scope type_scope function_scope _ seq_scope]
```

Elsewhere, as expected because of the *some-accumulator-is-decreasing requirement*, this attempt is not memorized by the « COQ » computer :

```
Fail
Fixpoint nat_empty (n : nat) {struct n}: False :=
  if n is S n' then nat_empty n' else nat_empty 0.
Fail Check nat_empty ( 3 : nat ) (** : False **).
```

```
The command has indeed failed with message:
Recursive definition of nat_empty is ill-formed.
In environment
nat_empty : nat -> False
n : nat
Recursive call to nat_empty has principal argument equal to
"0" instead of a subterm of "n".
Recursive definition is:
"fun n : nat =>
 match n with
 | 0 => nat_empty 0
 | succn n' => nat_empty n'
 end".

The command has indeed failed with message:
The reference nat_empty was not found in the current environment.
```

where « False » is the empty class / nat. Indeed « nat_empty : nat -> False » would say that the class « nat » is *empty* when in reality the class « nat » does contain the data element « 3 ».

3 Parametrism : parametric data, parametric functions

3.1 Option data

Now oneself wants some version of the data type « bool » which is container for more information / data, which is that each alternative « true | false » may contain more data which says *how* true (or how false).

Suppose oneself wants to write this partial function over only the *odd* numbers :

```
Definition pred_for_only_odd (n : nat) := if odd n then Some (n.-1) else None.
```

or suppose oneself wants to write such partial function over only the *small* numbers :

```
Definition odd_for_only_small (n : nat) := if n < 100 then Some (odd n) else None.
```

Therefore one may define some outer parametric / polymorphic data type as such :

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

```
About option.
About None.
About Some.
```

```
option : Type -> Type

option is template universe polymorphic
Argument scope is [type_scope]
Expands to: Inductive Coq.Init.Datatypes.option

None : forall A : Type, option A

None is template universe polymorphic
Argument A is implicit and maximally inserted
Argument scope is [type_scope]
Expands to: Constructor Coq.Init.Datatypes.None

Some : forall A : Type, A -> option A

Some is template universe polymorphic
Argument A is implicit and maximally inserted
Argument scopes are [type_scope _]
Expands to: Constructor Coq.Init.Datatypes.Some
```

The parameter « A » says that some different type / class « (option A) » exists for each possible choice of some type / class « A » , for example « (option nat) » , « (option bool) » … And « Type » is some keytext / keyword that denotes the class of all data classes, and « option » is some class / type former function, itself of class « (Type -> Type) ». Indeed « option » alone is not some data-type, but if oneself instantiates it with another data type, then it forms one. For example « (nat : Type) » and « (bool : Type) » are of class « Type », and may be used in place of « A » to produce the types « ((option nat) : Type) » and « ((option bool) : Type) ».

```
Check pred_for_only_odd : nat -> option nat.
Check odd_for_only_small : nat -> option bool.
```

Memo that all the constructors of this parametric / polymorphic data type definition, in reality, have some type parameter, where as described in some section above, the *classifying inference* keytext « forall .. , .. » is some more general form of the *inference* keytext « .. -> .. » .

The message « Argument A is implicit and ... » says that every time programmers write « Some » or « None » , the « COQ » computer automatically inserts / instantiates some term in place of the parameter « A » , such that this term does not lack to be textually written : the parameter is hidden / implicit. And the « COQ » computer infers or guesses whatever-is this type parameter

- when looking at the first explicit argument input given to « Some » , or
- when looking at the context surrounding the ouput of « None » .

```
Check Some 2.
```

```
Some 2
     : option nat
```

```
Check if (37 + 73) < 100 then Some (37+73) else None.
Eval compute in if (37 + 73) < 100 then Some (odd (37 + 73)) else None.
Fail Check if (37 + 73) < 100 then Some (odd (37 + 73)) else (None (A := nat)).
Fail Check if (37 + 73) < 100 then Some (odd (37 + 73)) else (@None nat).
Eval compute in if (37 + 73) < 100 then Some (odd (37 + 73)) else @None _.
```

```
if 37 + 73 < 100 then Some (37 + 73) else None
     : option nat

     = None
     : option bool

The command has indeed failed with message:
The term "None" has type "option nat" while it is expected to have type
 "option bool".

The command has indeed failed with message:
The term "None" has type "option nat" while it is expected to have type
 "option bool".

     = None
     : option bool
```

This example shows that the grammar « (None (A := nat)) » or « (@None nat) » may be used to force or input explicitly some parameter which is hidden, and that the grammar « _ » (underline, filter, hole, wildcard) as in « @None _ » may be used to explicitly-command « COQ » to attempt to infer some instantiation of any (implicit or explicit) parameter « A », for many reasons … In reality, the message « Argument A ... and maximally inserted » says that whenever the programmer writes « None » , the « COQ » computer translates it to « @None _ ».

3.2 List data

Now oneself wants some version of the data type « nat » which is container for more information / data, which is that each alternative « O | S (n : nat) » may contain more data which says *how* successor (or how zero). This is precisely the « parametric / polymorphic list / seq type » :

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

This parametric data type may indeed be instantiated to contain booleans « (seq bool) » or be instantiated to contain numbers « (seq nat) » :

```
Check cons true (cons false (cons true nil)).
Check cons 2 (cons 1 (cons 3 nil)).
Check 2 :: 1 :: 3 :: nil.
Check [:: 2; 1; 3].
Check fun l : seq nat => [:: 2, 1, 3 & l].
```

```
[:: true; false; true]
     : seq bool

[:: 2; 1; 3]
     : seq nat

[:: 2; 1; 3]
     : seq nat

[:: 2; 1; 3]
     : seq nat

fun l : seq nat => [:: 2, 1, 3 & l]
     : seq nat -> seq nat
```

In addition to the common functions which *precisely-inspect* the input, oneself may write parametric / polymorphic functions which *only-touch the form* of the input. For example oneself may write one single function which compute the number form (size) of any list (booleans list or numbers list or else) :

```
Fixpoint number A (s : seq A) : nat :=
  match s with
      cons _ tl => S (number tl)
    | nil => 0
  end.
```

where the parameter « A » of the function « number » is automatically memorized by « COQ » as hidden / implicit.

Elsewhere, memo that any sequence « [:: true; false; true] » may be commonly viewed as some function « [:: 0 |-> true; 1 |-> false; 2 |-> true] » which when given some position « 0 » , « 1 » or « 2 » output the corresponding item « true » , « false » or « true » of the sequence. Therefore oneself may input any sequence and input another function as both arguments to some composition / map (parametric) function :

```
Fixpoint map (A : Type) (B : Type) (f : A -> B) (s : seq A) : seq B :=
  if s is e :: tl
  then f e :: map f tl
  else nil.
```

For example to negate each item of some boolean list or to increment each item of some numbers list :

```
Eval compute in map (fun i : bool => ~~ i) [:: true; false; true].
Eval compute in map (fun i : nat => i.+1) [:: 2; 1; 3].
Eval compute in [seq i.+1 | i <- [:: 2; 1; 3]].
```

```
= [:: false; true; false]
: seq bool

= [:: 3; 2; 4]
```

```
: seq nat

= [:: 3; 2; 4]
: seq nat
```

where this more-advanced notation which binds / holds some identifier / name, was used (the identifier « i » is bound / held in the term « E ») :

```
Notation "[ 'seq' E | i <- s ]" := (map (fun i => E) s).
```

```
Reset map.
```

In addition to the function « map » there are more *advanced-parametric* function which only-touch sequences, as described in the file « seq.v ». For instance the « filter » function and its notation « [seq i <- s | p] » filters the sequence « s » keeping only the values selected by the boolean test « p », whenever the common parameter « A » itself-as-data may be classified ("canonical structures") …

Elsewhere, memo that because « map » takes *some function as input* it is said to be some higher-order function. Morever « map » may also be viewed as *outputing some function* when « map » is only partially applied / inputed :

```
Check map (fun i : bool => ~~ i) [:: true; false; true].
Check map (fun i : bool => ~~ i) .
```

```
[seq ~~ i | i <- [:: true; false; true]]
     : seq bool

map [eta negb]
     : seq bool -> seq bool
```

Finally, oneself shall clarify the link of « bool » with « option », and the link of « nat » with « seq ». Primo, define the (non-parametric) unit type / class which has some single constructor « tt » (and only one inhabitant) :

```
Inductive unit : Type :=  tt : unit.
```

```
Reset unit.
```

Secondo, it is now clear that there is some correspondence of « bool » with « option unit », and that there is some correspondence of « nat » with « seq unit » …

4 Classification

Another review at this instance of recursive function defined over the numbers

```
Print nat_rect.
```

```
nat_rect =
fun (P : nat -> Type) (f : P 0) (f0 : forall n : nat, P n -> P (succn n)) =>
```

```
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | succn n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (succn n)) -> forall n : nat, P n

Argument scopes are [function_scope _ function_scope nat_scope]
```

shows that these given alternate constructions « O | S (n : nat) » of the class « nat » computationally or logically fulfill / support this class, which is that it is sufficient to focus / touch on these (recursively) constructored data when holding this class, which is that any (random) data in the class may be such (recursively) destructed / eliminated / matched / filtered. This is described by this (grammatical) destructor / match function

```
match .. as .. return .. with .. => .. | .. => .. end
```

and maybe, for enabling recursion / reference to self (here « F » may textually mention « F » in its definition), some surrounding « fix F n » keytext instead of the « fun n » keytext :

```
fix .. := ..
```

Now while any « bool » or « nat » or « list nat » or « list A » is some single isolated class / type, this above expression « (P : nat -> Type) » says that « P » is some classification or some family / collection of classes or some dependent class or some predicate; such that « P 0 : Type » is some class and « P 1 : Type » is another class and « P 2 : Type » is also some class … And oneself is attempting to define some (recursive) function « nat_rect » from the class « nat » to the classification « P », such that the input « n » from « nat » *affect both* the precise-classification of the output (here « P n ») and the value of the output (here « f » when « n » is « O » , or « f0 n0 (F n0) » when « n » is « S n0 »).

Saying how the input affect the *precise-classification* of the output is done by the keytext

```
as .. return ..
```

Saying how the input affect the *value* of the output is done by the keytext

```
with .. => .. | .. => .. end
```

.. and certainly each filtering of the input for deciding which alternative branching will, as usual, instantiate / refine the proposed output value but also instantiate / refine the expected classification / type of the output.

4.1 List classification

Another example of some classification similar as « (P : nat -> Type) » is the indexed lists, which futher classifies lists by some index / argument, whose sense here is similar as « size (number form) of the list » :

```
Inductive ilist : nat -> Type :=
    inil : ilist 0
  | icons : nat -> forall m : nat, ilist m -> ilist (S m).
```

```
ilist is defined
ilist_rect is defined
ilist_ind is defined
ilist_rec is defined
```

Attempt now to define some (recursive) function from « nat » to « ilist » , which input some number « m » and output something precisely-classified by « ilist m » and whose value has all its items being « 7 » :

```
Definition only7 :=
  fix only7 (m : nat) : (ilist m) :=
    match m as m' return (ilist m') with
      O => inil
    | S m0 => icons 7 (only7 m0)
    end .
```

```
Eval compute in only7 3.
```

```
= icons 7 (icons 7 (icons 7 inil))
: ilist 3
```

Another way of writing the same thing is :

```
Reset only7.
Definition only7 :=
  nat_rect (ilist) (inil) (fun (m0 : nat) (only7_m0 : ilist m0) => icons 7 only7_m0).
```

4.2 Equality classification

Another example of some classification similar as « (P : nat -> Type) » is the equality classification, which classifies (primitive or complex) deductions / proofs of "equality" to some given fixed (outer) parameter « x » ; this classification is by whatelse (inner) arguments / indices this fixed parameter « x » is possibly (by-computation-and-by-logical-deduction-)"equal" (eq) to :

```
Inductive eq (A : Type) (x : A) : A -> Type :=
  eqrefl : @eq A x x .
```

```
eq is defined
eq_rect is defined
eq_ind is defined
eq_rec is defined
```

```
About eqrefl.
```

```
eqrefl : forall (A : Type) (x : A), eq x x

eqrefl is template universe polymorphic
Argument A is implicit
Argument scopes are [type_scope _]
```

```
Expands to: Constructor Top.eqrefl
```

And the infix notation « x = a » is often used for saying « @eq A x a », where the hidden / implicit parameter « A » will be inferred / guessed by the « COQ » computer.

From the meta (outer) angle of view, « eqrefl » is some primitive deduction / proof term / value which says (asserts / deduces / proves) that the parameter « x » is "eq" ("equal") to any index-argument « y » which by-computation-is (by-convertibility-is) « x » ; and « eqrefl » is precisely-classified inside « @eq A x y » by any index-argument « y » which by-computation-is « x ».

```
Check @eqrefl nat (3 + 2) : @eq nat (3 + 2) (S (S (1 + 2))).
```

```
eqrefl (3 + 2) : eq (3 + 2) (succn (succn (1 + 2)))
     : eq (3 + 2) (succn (succn (1 + 2)))
```

From the meta (outer) angle of view, the description of "eq" above may be read as :

```
Inductive eq (A : Type) (x : A) : A -> Type :=
  eqrefl : forall ?y which by-computation-is x , @eq A x ?y .
```

Now, deductions may be combinated in complex ways; which is that oneself may arrive at « @eq A x y » by more-complex combinaisons (by-logical-deduction) beyond precisely-one-primitive-deduction (by-computation) « eqrefl » . Therefore any (random) data / inhabitant « lemma1 : @eq A x z » is named logical-deduction.

Here is one instance of combining deductions : how to input some deduction data from « (@eq nat x a) » and output some deduction into « (@eq nat (S x) (S a)) » which is further-classified by the index-argument « (S a) » of the classification « (@eq nat (S x)) » . This uses the (grammatical) destructor / match function on "equality", which is named (congruent) rewriting / casting / transport. This may look silly :

```
Check
  (fun (x : nat) (a : nat) (H : @eq nat x a) =>
     match H as H0 in (@eq _ _ a0) return (@eq nat (S x) (S a0)) with
     | @eqrefl _ _ => @eqrefl nat (S x)
     end)
     : forall x a : nat, @eq nat x a -> @eq nat (S x) (S a) .
```

```
(fun (x a : nat) (H : eq x a) =>
 match H in (eq _ a0) return (eq (succn x) (succn a0)) with
 | @eqrefl _ _ => eqrefl (succn x)
 end)
  : (forall x a : nat, eq x a -> eq (succn x) (succn a))
     : forall x a : nat, eq x a -> eq (succn x) (succn a)
```

Saying how the input affect the precise-classification of the output is done by the keytext

```
as .. in .. return ..
```

which has some extra « in .. » keytext now because the input data is taken from some *classification* « (@eq nat (S x)) » instead of from only some class such as the earlier « bool » or « nat » examples.

This full grammar « as .. in .. return .. » is because « the input » « H » is both the input value (the « H0 » in the « as H0 » keytext) and the input precise-classification (the argument « a0 » in the « in @eq _ _ a0 » keytext); and both the input value and input precise-classification do affect the expected output precise-classification (here « return (@eq nat (S x) (S a0)) » , where « H0 » is absent here only-by-chance). However the value of the output in each alternative branch has only access to the destructed value of the input and *no direct-access* to the precise-classification of the input.

Another example which uses this full grammar « as .. in .. return .. » is this « more » function, which increments all the items in some indexed list and creates one more item :

```
Definition more :=
  (fix more (m : nat) (l : ilist m) {struct l} : ilist (S m) :=
     match l as l0 in ilist m0 return ilist (S m0) with
       inil => icons 0 inil
     | icons j m_tl tl => icons (S j) (more m_tl tl)
     end).
```

```
Eval compute in @more 3 (icons 6 (icons 7 (icons 5 inil))).
```

```
= icons 7 (icons 8 (icons 6 (icons 0 inil)))
: ilist 4
```

Another example which uses nothing of this full grammar « as .. in .. return .. » (because the ouput classification « nat » does not lack it) is this « inumber » ("isize") function, which computes the *real size* (number of items, number form) of some indexed list.

```
Definition inumber :=
  (fix inumber (m : nat) (l : ilist m) {struct l} : nat :=
     match l with
       inil => 0
     | icons j m_tl tl => S (inumber m_tl tl)
     end).
```

Finally one may (recursively) program some deduction « lemma1 » which deduces that the (sense of the) index-argument (the precise-classification) of some indexed list is indeed "equal" (eq) to the real size (number of items) of this indexed list.

```
Definition lemma1 :=
  fix lemma1 (m : nat) (l : ilist m) {struct l} : @eq nat m (inumber l) :=
    match l as l0 in ilist m0 return @eq nat m0 (inumber l0) with

      inil =>
        (** expected (goal) output precise-classification :
        @eq nat 0 (inumber (inil)) , which computationally-is
        @eq nat 0 O  **)
        @eqrefl nat 0

    | icons j m_tl tl =>
        (** expected (goal) output precise-classification :
        @eq nat (S m_tl) (inumber (icons j m_tl tl)) , which computationally-is
        @eq nat (S m_tl) (S (inumber tl)) ;

        now by recursion, the deduction (lemma1 m_tl tl) of classification
        @eq nat (m_tl) ((inumber tl)),
        is present to do some rewrite / cast / transport  **)
        match (lemma1 m_tl tl) as H0 in @eq _ _ a0 return @eq nat (S m_tl) (S a0) with
          @eqrefl _ _ =>
            (** expected (goal) output precise-classification :
            @eq nat (S m_tl) (S m_tl)  **)
            @eqrefl nat (S m_tl)
        end

    end.
```

Such programming of deduction may also be done using script / tactical commands which prevent programmer fatigue and sometimes automate the writing of very long deductions / proof terms / values. Such script is started by the « Lemma » command, then the « COQ » computer prints the expected (goal) classification of the ouput deduction :

```
Lemma lemma2 : forall (m : nat) (l : ilist m), @eq nat m (inumber l).
```

```
1 subgoal

  ============================
  forall (m : nat) (l : ilist m), eq m (inumber l)
```

Early oneself may decide-and-rest the names of the quantified variables or assumptions, by using the « intros » command (or the almost-same « move => .. » command) :

```
(** move => m l .**)
intros m l.
```

```
1 subgoal

  m : nat
  l : ilist m
  ============================
  eq m (inumber l)
```

Now the « fix lemma2 m l {struct l} := match l .. with .. => .. | .. => .. end » keytext corresponds to the « induction » command (or the almost-same « elim » command), then the « COQ » computer prints, for each filtering of the input, the expected (goal) precise-classification of the output deduction; the « inil => .. » alternative branch shall be solved first :

```
(** elim : l => [ | j m_tl tl lemma2_m_tl_tl ] .**)
induction l as [ | j m_tl tl lemma2_m_tl_tl ]  .
```

```
2 subgoals

  ============================
  eq 0 (inumber inil)

subgoal 2 is:
 eq (succn m_tl) (inumber (icons j tl))
```

Now the (hidden, implicit, automatic) computation by the « COQ » computer in this branch may be made explicit by using the « simpl » command (or the almost-same « .. => /= .. » command).

```
(** move => /= .**)
simpl.
```

```
2 subgoals

  ============================
  eq 0 0

subgoal 2 is:
 eq (succn m_tl) (inumber (icons j tl))
```

Finally for this branch, the basic (constructor) deduction « @eqrefl nat 0 » is *exactly / precisely* what will solve this goal and print the next goal, therefore oneself may use the « exact » command :

```
exact (@eqrefl nat 0).
```

```
1 subgoal

  j, m_tl : nat
  tl : ilist m_tl
  lemma2_m_tl_tl : eq m_tl (inumber tl)
  ============================
  eq (succn m_tl) (inumber (icons j tl))
```

Again the (hidden, implicit, automatic) computation by the « COQ » computer in this second branch « icons j m_tl tl => .. » may be made explicit by using the « simpl » command.

```
(** move => /= .**)
simpl.
```

```
1 subgoal

  j, m_tl : nat
  tl : ilist m_tl
  lemma2_m_tl_tl : eq m_tl (inumber tl)
  ============================
  eq (succn m_tl) (succn (inumber tl))
```

Now the rewriting / cast / transport is done in 3 progress; the first progress, which is the « revert » command (or the almost-same « move : .. » command) followed by the « generalize » command (or the almost-same « move : .. » command), corresponds to the keytext « (lemma2_m_tl_tl) as H0 in @eq _ _ a0 return @eq nat (S m_tl) (S a0) » :

```
(** move : lemma2_m_tl_tl .**)
revert lemma2_m_tl_tl.
```

```
1 subgoal

  j, m_tl : nat
  tl : ilist m_tl
  ============================
  eq m_tl (inumber tl) -> eq (succn m_tl) (succn (inumber tl))
```

```
(** move : (inumber tl) .**)
generalize (inumber tl) as a0.
```

```
1 subgoal

  j, m_tl : nat
  tl : ilist m_tl
  ============================
  forall a0 : nat, eq m_tl a0 -> eq (succn m_tl) (succn a0)
```

These two related steps of the same progress may be combined by the « .. ; .. » command (or may be combined on the same text line of the « .. : .. » command) :

```
Undo 2.
```

```
(** move : (inumber tl) lemma2_m_tl_tl .**)
revert lemma2_m_tl_tl ; generalize (inumber tl) as a0 .
```

And the second progress which is the « destruct » command (or the almost-same « case » command), corresponds to the keytext « match .. with @eqrefl _ _ => .. end » :

```
(** move => a0 ; case .**)
destruct 1.
```

```
1 subgoal

  j, m_tl : nat
  tl : ilist m_tl
  ============================
  eq (succn m_tl) (succn m_tl)
```

Memo that if this hand-crafted « eq » were the initial real « COQ » « eq », then these 2 progresses (« revert » followed by « generalize » followed by « destruct ») may be combined into some single command « rewrite <- lemma2_m_tl_tl . » , where the sense of the arrow « <- » is *right-to-left rewriting*.

Finally for this branch, the basic (constructor) deduction « @eqrefl nat 0 » is *exactly / precisely* what will make the third progress and solve this goal. Alternatively of using the « exact (@eq_refl nat (S m_tl)). » command, oneself may use the « apply » command, which allow to progress slowly partially, eventually producing some rest of goals (here none) :

```
(** apply : eqrefl .**)
apply eqrefl.
```

```
No more subgoals.
```

And these next commands tell the « COQ » computer to memorize the deduction (as « lemma2

»), and then to print its deduction term / value for comparison with the manually-programmed deduction term memorized in « lemma1 ».

```
(** Qed .**)
Defined.
```

```
(intros m l).
(induction l as [| j m_tl tl lemma2_m_tl_tl]).
 (simpl).
 exact (@eqrefl nat 0).

 (simpl).
 (revert lemma2_m_tl_tl; generalize (inumber tl) as a0).
 (destruct 1).
 (apply eqrefl).

Defined.
lemma2 is defined
```

```
Print lemma2.
```

```
lemma2 =
fun m : nat =>
[eta ilist_ind (eqrefl 0)
       (fun (j m_tl : nat) (tl : ilist m_tl) =>
        [eta (fun (a0 : nat) (lemma2_m_tl_tl0 : eq m_tl a0) =>
              match
                lemma2_m_tl_tl0 in (eq _ y)
                return (eq (succn m_tl) (succn y))
              with
              | @eqrefl _ _ => eqrefl (succn m_tl)
              end) (inumber tl)]) (n:=m)]
     : forall (m : nat) (l : ilist m), eq m (inumber l)

Argument m is implicit
Argument scopes are [nat_scope _]
```

```
Eval unfold lemma2, ilist_ind, ilist_rect in lemma2.
```

```
= fun (m : nat) (l : ilist m) =>
  (fix F (n : nat) (i : ilist n) {struct i} :
   eq n (inumber i) :=
     match i as i0 in (ilist n0) return (eq n0 (inumber i0)) with
     | inil => eqrefl 0
     | @icons _ m0 i0 =>
         match F m0 i0 in (eq _ y) return (eq (succn m0) (succn y)) with
         | @eqrefl _ _ => eqrefl (succn m0)
         end
     end) m l
: forall (m : nat) (l : ilist m), eq m (inumber l)
```

5 Deduction : functions

5.1 Goal as nested-stack

Primo, the angle of view is that any class / goal such as « ((nat -> nat) -> nat -> nat) » is similar as some nested stack.

Secondo, memo that there is some flexibility when writing some type / goal, as in :

```
Goal forall xy : prod nat nat, prime (fst (xy : (nat * nat)%type)) ->
                 odd xy.2 = true -> leq 2 ((snd xy) + xy.1) .
```

```
1 subgoal

  ============================
  forall xy : nat * nat,
  prime (xy : nat * nat).1 -> odd xy.2 = true -> 1 < xy.2 + xy.1
```

```
Unset Printing Notations.
Show.
```

```
1 subgoal

  ============================
  forall (xy : prod nat nat) (_ : prime (fst (xy : prod nat nat)))
    (_ : Logic.eq (odd (snd xy)) true),
  leq (S (S O)) (addn (snd xy) (fst xy))
```

Here the pairing type former « prod » (described below) is subtituted by its infix notation « * » , which may be precised by some more annotation / scope « (..)%type » when it is necessary to distinguish it from the infix notation « * » of « multn » , which may be precised by some other annotation / scope « (..)%nat » . Also « .1 » and « .2 » are postfix notations for correspondingly the functions « fst » and « snd » .

```
Locate "*".
```

```
Notation            Scope
"m * n" := Nat.mul m n         : coq_nat_scope

"m * n" := muln_rec m n        : nat_rec_scope

"m * n" := muln m n  : nat_scope
                      (default interpretation)
"x * y" := prod x y  : type_scope
```

Elsewhere one may write « leq 2 ((snd xy) + xy.1) » , instead of « leq 2 ((snd xy) + xy.1) = true » as it was done for « odd xy.2 = true » . Similarly one may write « prime xy.1 » , instead of « prime xy.1 = true » . This notational process is named « coercive notation » (or « automatic declassification ») and is based from this function :

```
Eval compute in (fun b : bool => (is_true b : Type)).
```

```
= fun b : bool => Logic.eq b true
: forall _ : bool, Type
```

```
Set Printing Coercions.
Show.
```

```
1 subgoal

  ============================
  forall (xy : prod nat nat) (_ : is_true (prime (fst (xy : prod nat nat))))
    (_ : Logic.eq (odd (snd xy)) true),
  is_true (leq (S (S O)) (addn (snd xy) (fst xy)))
```

5.2 Intro from, apply in, specialize of, substitution by - the nested-stack

The postfix « => ... » grammar may be used as post-processing phase of any proof command, and it does some sequence of actions on the now-present top / first assumption or quantified variable in the goal.

```
Goal forall xy : nat * nat, prime xy.1 -> odd xy.2 -> 2 < xy.2 + xy.1 .
```

```
1 subgoal

  ============================
  forall xy : nat * nat, prime xy.1 -> odd xy.2 -> 2 < xy.2 + xy.1
```

```
move => xy .
```

```
1 subgoal

  xy : nat * nat
  ============================
  prime xy.1 -> odd xy.2 -> 2 < xy.2 + xy.1
```

```
Undo.
```

```
move => xy => pr_x odd_y .
```

```
1 subgoal

  xy : nat * nat
  pr_x : prime xy.1
  odd_y : odd xy.2
  ============================
  2 < xy.2 + xy.1
```

where the « move. » command alone without post-processing does nothing (almost, as in reality it performs *head normal form computation* such to expose any prefix « forall » classifying inference / quantification). And the postfix « => .. » grammar introduces the top assumption or variable into the (outer) context.

Now, en passant, oneself may want to decompose « xy » into its first and second component. Instead of the long text « move=> xy; destruct xy as [x y] » , oneself may use this shorter grammar « [] » for the introduction filter to perform such action.

```
move=> [x y] pr_x odd_y.
```

```
1 subgoal

  x, y : nat
  pr_x : prime (x, y).1
  odd_y : odd (x, y).2
  ============================
  2 < (x, y).2 + (x, y).1
```

Oneself may place the « /= » option to command « COQ » to partially-compute (simplify) the terms on the nested-stack before introducing them into the (outer) context. This is the same as the « simpl. » command.

```
move=> [x y] /= pr_x odd_y.
```

```
1 subgoal

  x, y : nat
  pr_x : prime x
  odd_y : odd y
  ============================
  2 < y + x
```

Oneself may also apply / instantiate some lemma by some assumption (or confusingly, *view some assumption*). For example the lemma « prime_gt1 : forall p : nat, prime p -> 1 < p » , where the variable argument « p » is implicit, may be be applied onto (instantiated by) the top assumption « (prime x) » such to transform the top assumption as now « (1 < x) », by placing the « /prime_gt1 » option (*view*) on the « => ... » grammar line. This is the same as the common « apply .. in .. » command …

```
move=> [x y] /= /prime_gt1-x_gt1 odd_y.
```

```
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  odd_y : odd y
  ============================
  2 < y + x
```

where the « - » text, which visually link the function and name assigned to its output, has no effect and may be erased.

In reality, the class / type of the applied lemma « lemma1 : A -> B » on the top assumption « A » of the inner context not lack to be of the precise form « A -> B » ; for example this lemma may be of type « lemma1 : B <-> A » and therefore the « COQ » computer shall automatically-transform this lemma from « (lemma1) » to « (@iffRL _ _ (lemma1)) : A -> B » before actually applying it onto the top assumption « A » of the inner context. This process is named enabling the query of the view / transformation hints; and some of these hints are already pre-memorized in the « ssreflect.v » file after some « Require Import mathcomp.ssreflect.ssreflect. » command.

```
Check @iffRL : forall P Q (eqPQ : P <-> Q), Q -> P.
```

```
Hint View for move/ iffRL|2 (** this 2 refer to the 2 _ prefixing @iffRL _ _ above**).
```

```
iffRL : (forall P Q : Prop, P <-> Q -> Q -> P)
     : forall P Q : Prop, P <-> Q -> Q -> P
```

```
move=> [x y] /= /prime_gt1'-[x_gt1 x_pr] odd_y.
```

```
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  x_pr : prime x
  odd_y : odd y
  ============================
  2 < y + x
```

Oneself may also examine « y » : it shall not be « 0 » , since this would make the assumption « odd y » compute to « false » which is some easy contradiction which immediately solves this branch of the goal (in other words : the zero case / branch where « y » is « 0 » is immediately solved because of the presence of some easy contradiction in the assumptions). The « // » option of the « => ... » grammar commands « COQ » to attempt to *immediately solve* the goal. This is the same as the « done. » or « by []. » composite-commands …

```
move=> [ x [ // | y ] ] /= /prime_gt1-x_gt1.
```

```
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  ============================
  ~~ odd y -> 2 < succn y + x
```

where the « [.. | ..] » grammar here is same as « destruct y as [| y] » which generates two branches / cases for the two constructors of the data class « nat » . Memo that the same name (here « y ») may be masking / reused for filter-held / filter-bound filter-variables, in some comparable way as for funtional-held function-variable.

Now, oneself knows that the assumption saying « y » is even is not lacked to solve, therefore oneself may clear this assumption by one of two ways : by first introducing it using some common name then clearing it using the « { .. } » option or by introducing it using the « _ » dummy name. Memo that the « by » solving / closing command (and its introduction-filter synonym « .. => // » command) attempts to immediately-solve any resting goal (here none) *orelse fails*, and is some visual textual marker to communicate that the now-present goal is being *fully resolved*.

```
move=> [x [//|y]] /= /prime_gt1-x_gt1 odd_y {odd_y}.
```

```
by move=> [ x [ // | y ] ] /= /prime_gt1-x_gt1 _ ; apply (ltn_addl _ x_gt1).
```

```
No more subgoals.
```

```
About ltn_addl.
```

```
ltn_addl : forall m n p : nat, m < n -> m < p + n

Arguments m, n are implicit
Argument scopes are [nat_scope nat_scope nat_scope _]
ltn_addl is opaque
Expands to: Constant mathcomp.ssreflect.ssrnat.ltn_addl
```

Dually of *apply / instantiate some lemma by some assumption*, oneself may also specialize/instantiate some assumption by some lemma :

```
Goal (forall n, n * 2 = n + n) -> 6 = 3 + 3.
```

```
1 subgoal

  ============================
  (forall n : nat, n * 2 = n + n) -> 6 = 3 + 3
```

```
move => /(_ 3).
```

```
1 subgoal

  ============================
  3 * 2 = 3 + 3 -> 6 = 3 + 3
```

This is almost same as applying / instantiating some other lemma « special1arg » by this same assumption.

```
move => /(special1arg 3).
```

```
1 subgoal

  ============================
  3 * 2 = 3 + 3 -> 6 = 3 + 3
```

Moreover, when the top stack item is some equation, oneself may substitute-rewrite by this equation then clear this equation. The options « <- » and « -> » for the « .. => ... » grammar correspond to right-to-left and left-to-right substitutions. This is the same as the common « rewrite » command, but now the equation is *cleared* from the assumptions or outer context.

```
move => <- .
```

```
1 subgoal

  ============================
  6 = 3 * 2
```

Finally, memo that the apply-in option (« /lemma1 ») and specialize option (« /(_ input1) ») of the « move => .. » command may occur affixed to the « move/ » keytext :

```
Undo 2.
move/(special1arg 3).
```

```
1 subgoal

  ============================
  (forall n : nat, n * 2 = n + n) -> 6 = 3 + 3

1 subgoal

  ============================
  3 * 2 = 3 + 3 -> 6 = 3 + 3
```

```
Undo 1.
move/(_ 3).
```

```
1 subgoal

  ============================
  (forall n : nat, n * 2 = n + n) -> 6 = 3 + 3

1 subgoal

  ============================
  3 * 2 = 3 + 3 -> 6 = 3 + 3
```

And memo for later that the command « case/lemma1 » or « case/(_ input1) » correspondingly-is short for the command « move/lemma1; case. » or « move/(_ input1); case. » .

5.3 Revert to, generalize in, unification of - the nested-stack

The postfix « .. : .. » grammar may be used as pre-processing phase of many proof commands such as « move », « case », « elim » (and « apply » …), and it essentially push assumptions onto the goal nested-stack from somewhere else …

5.3.1 revert

Oneself may want to de-structure « y » at this present-time, but oneself lacks to pre-process the goal :

```
Lemma goal1 (x y : nat) (x_gt1 : 1 < x) (odd_y : odd y) : 2 < y + x .
```

```
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  odd_y : odd y
  ============================
  2 < y + x
```

```
Fail move : y .
move : y odd_y .
```

```
The command has indeed failed with message:
Ltac call to "move (ssrmovearg) (ssrclauses)" failed.
Error: y is used in hypothesis odd_y.
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  odd_y : odd y
  ============================
  2 < y + x

1 subgoal

  x : nat
  x_gt1 : 1 < x
  ============================
  forall y : nat, odd y -> 2 < y + x
```

Here the assumptions are pushed onto the goal nested-stack *from the outer context*; this process is named reverting variables / hypothesis.

Then one more line containing the « case. » command will make the lacked progress. Alternatively oneself may combine the « .. : .. » pre-processing grammar with the « case » (destruct) processing with the « .. => .. » post-processing grammar on one single line of text.

```
case : y odd_y => [ | y' ].
```

```
2 subgoals

  x : nat
  x_gt1 : 1 < x
  ============================
  odd 0 -> 2 < 0 + x

subgoal 2 is:
 odd (succn y') -> 2 < succn y' + x
```

which is same as commanding « move : y odd_y ; case => [| y']. » , where exceptionally-for-case the « case => [.. | ..] » grammar with the bracketing « [.. | ..] » immediately after the arrow « => » signify *branching* instead of (one more) *destruction* … When oneself really wants some additional *destruction*, one shall write « case => - [.. | ..] » where the « - » keytext has no other effect.

```
Admitted.
```

```
goal1 is declared
```

5.3.2 generalize, unification-generalize

Such above goal may sometimes occur in less-practical form :

```
Lemma goal2 (x y1 y2 : nat) (x_gt1 : 1 < x)
   (odd_y1y2 : odd (y1 - y2)) : 2 < (y1 - y2) + x .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  2 < y1 - y2 + x
```

and therefore the term « (y1 - y2) » may be generalized such that oneself is back to the more-practical form above and oneself may continue the same deduction :

```
move : (y1 - y2) odd_y1y2 .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  ============================
  forall subn : nat, odd subn -> 2 < subn + x
```

Alternatively, oneself may save / memorize this rest of the same deduction as some new lemma « goal1 » (this was done above), then immediately « apply » this saved lemma at the start of this new deduction instead of manually-generalizing. This senses that the « apply » command is some unification-generalize command.

```
Undo.
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  2 < y1 - y2 + x
```

```
apply (goal1) .
```

```
2 subgoals

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  1 < x

subgoal 2 is:
 odd (y1 - y2)
```

Yet another re-wording of this instantiation / application of lemma « goal1 » is as follows :

- primo, oneself pushes the assumption which is the class / type of the lemma « goal1 » on top of the now-present inner-context goal, which is that oneself *generalizes the now-present goal by any deduction of the class / type* of the lemma « goal1 » , instead of directly-using the particular lemma « goal1 » ,
- secondo, in the inner-context, oneself apply this top assumption onto the conclusion.

```
move : goal1 .
apply .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  (forall x0 y : nat, 1 < x0 -> odd y -> 2 < y + x0) -> 2 < y1 - y2 + x

2 subgoals

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  1 < x

subgoal 2 is:
 odd (y1 - y2)
```

In reality, the following command will combine these two steps :

```
apply : goal1 .
```

```
2 focused subgoals
(shelved: 2)

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  1 < x

subgoal 2 is:
 odd (y1 - y2)
```

and is precisely-same as these scripted composite-commands, where the « cmd1 || cmd2 » command says that « cmd1 » succeed-and-progress else « cmd2 » :

```
refine ( @goal1 ) || ( refine ( @goal1 _ ) ||
( refine ( @goal1 _ _ ) || ( refine ( @goal1 _ _ _ ) ||
refine ( @goal1 _ _ _ _ ) ) ) ) (** or more **) .
```

```
2 subgoals
```

```
  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  1 < x

subgoal 2 is:
 odd (y1 - y2)
```

Moreover, oneself may enable the querying of the view / transformation hints database by affixing the lemma-to-be-applied to the « apply/ » command :

```
apply / goal1 .
```

```
2 focused subgoals
(shelved: 2)

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  1 < x

subgoal 2 is:
 odd (y1 - y2)
```

Finally, memo that the lemma may be some variable / hypothesis name in the outer context instead of some save / memorized top / global lemma, in which case oneself shall surround the variable / hypothesis name by parenthesis such to prevent such name from being *cleared* from the outer context. For example :

```
move : (x_gt1) .
```

```
2 focused subgoals
(shelved: 1)

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  1 < x -> 1 < x

subgoal 2 is:
 odd (y1 - y2)
```

```
Abort.
```

5.3.3 revert-then-intro

Here is the process named revert-then-intro (contextualization of commands). For the same goal « goal2 » as above,

```
Lemma goal2 (x y1 y2 : nat) (x_gt1 : 1 < x)
   (odd_y1y2 : odd (y1 - y2)) : 2 < (y1 - y2) + x .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  2 < y1 - y2 + x
```

when oneself wants to immediately re-introduce into the outer context all the changed hypotheses (of the *outer context*) which are now assumptions (of the *inner context*) of the generalized goal, oneself may do this command :

```
move : (y1 - y2) => z in odd_y1y2 * .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  z : nat
  odd_y1y2 : odd z
  ============================
  2 < z + x
```

instead of this less-clear command :

```
move : (y1 - y2) odd_y1y2 => z odd_y1y1 .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  z : nat
  odd_y1y1 : odd z
  ============================
  2 < z + x
```

where the optional « * » keytext communicates that the inner-context of the goal is also affected by the tactic command, and not only the hypotheses explicitly selected.

Moreover memo that the no-longer-lacked variables « y1 » and « y2 » may be cleared by using the braces « { .. } » option in the same line of textual command :

```
move : (y1 - y2) => {y1 y2} z in odd_y1y2 * .
```

```
1 subgoal

  x : nat
  x_gt1 : 1 < x
  z : nat
  odd_y1y2 : odd z
  ============================
  2 < z + x
```

5.3.4 equational-generalize

For the same goal « goal1 » as above, oneself may describe some name for some equation which links the term at the top of the stack before and after the de-structuring command. This process is named equational-generalize.

```
Lemma goal3 (x y : nat) (x_gt1 : 1 < x) : odd y -> 2 < y + x .
```

```
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  ============================
  odd y -> 2 < y + x
```

```
move : {-1}y (erefl y). (** equational-generalize **)
case => [ | y' ] E (** case **).
```

```
1 subgoal

  x, y : nat
  x_gt1 : 1 < x
  ============================
  forall y0 : nat, y = y0 -> odd y0 -> 2 < y0 + x

2 subgoals

  x, y : nat
  x_gt1 : 1 < x
  E : y = 0
  ============================
  odd 0 -> 2 < 0 + x

subgoal 2 is:
 odd (succn y') -> 2 < succn y' + x
```

where the braces « { .. } » option in « {-1}y » select which occurrences of the term « y » shall be generalized or not generalized.

Alternatively, oneself may use the equation-option of the case command.

```
case E : y => [ | y' ] (** equational-generalize case **).
```

```
2 subgoals

  x, y : nat
  x_gt1 : 1 < x
  E : y = 0
  ============================
  forall y0 : nat, 0 = y0 -> odd y0 -> 2 < y0 + x

subgoal 2 is:
 forall y0 : nat, succn y' = y0 -> odd y0 -> 2 < y0 + x
```

```
Abort.
```

5.3.5 lessorequal-generalize

For deduction by induction / recursion, during the induction step at « (S n) » sometimes it is necessary to apply the induction hypothesis :

- not on the immediate-predecessor « (S n) - 1 » which is « n », as commonly done,
- but on some deeper-predecessor such as « (S n) - 4 » which is « (n - 3) ».

Therefore oneself shall pre-process the goal by doing what is named equational-generalize, and then do the induction « elim » command such to get what is named lessorequal-generalize induction (or strong induction) . This arithmetic question is some instance :

```
Lemma stamps n : 12 <= n -> exists s4 s5, s4 * 4 + s5 * 5 = n.
```

```
1 subgoal

  n : nat
  ============================
  11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n
```

```
move : n {-2}n (leqnn n). (** lessorequal-generalize **)
elim => [ | m IHm ] (** elim , then [  |  ] branching and intros **) .
Show 2 (** shows subgoal 2 which has the stronger induction hypothesis **).
```

```
1 subgoal

  ============================
  forall n n0 : nat,
  n0 <= n -> 11 < n0 -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n0

2 subgoals

  ============================
  forall n : nat, n <= 0 -> 11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n

subgoal 2 is:
 forall n : nat,
 n <= succn m -> 11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n

subgoal 2 is:

  m : nat
  IHm : forall n : nat,
        n <= m -> 11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n
  ============================
  forall n : nat,
  n <= succn m -> 11 < n -> exists s4 s5 : nat, s4 * 4 + s5 * 5 = n
```

```
Abort.
```

5.3.6 initial-accumulator-generalize

Another *embrouille* when attempting to do deduction by induction immediately / suddenly / non-slowly is that some occurrences of initial-memories / initial-accumulators in the goal may lack to be generalized earlier before doing the actual induction later, because the value of the accumulator memory is going to *change* when effecting the recursive calls and therefore the induction hypothesis shall be general. This mediating pre-processing is named initial-accumulator-generalize.

```
Fixpoint addacc (n : nat) (a : nat) {struct n} : nat :=
  if n is S n' then addacc n' (S a) else a .
Definition add10 (n : nat) := addacc n 10 .
```

```
addacc is defined
addacc is recursively defined (decreasing on 1st argument)

add10 is defined
```

```
Lemma add10S (n : nat) : add10 (S n) = S (add10 n).
rewrite /add10.
```

```
1 subgoal

  n : nat
  ============================
  add10 (succn n) = succn (add10 n)

1 subgoal

  n : nat
  ============================
  addacc (succn n) 10 = succn (addacc n 10)
```

```
move : 10 .
```

```
1 subgoal

  n : nat
  ============================
  forall n0 : nat, addacc (succn n) n0 = succn (addacc n n0)
```

```
elim : n => [ // | n IHn ] a .
```

```
1 subgoal

  n : nat
  IHn : forall n0 : nat, addacc (succn n) n0 = succn (addacc n n0)
  a : nat
  ============================
  addacc (succn (succn n)) a = succn (addacc (succn n) a)
```

```
simpl.
apply: (IHn (S a)).
```

```
1 subgoal
```

```
  n : nat
  IHn : forall n0 : nat, addacc (succn n) n0 = succn (addacc n n0)
  a : nat
  ============================
  addacc n (succn (succn a)) = succn (addacc n (succn a))

No more subgoals.
```

```
Qed.
```

5.3.7 forward-generalize, backward-generalize

In the section << generalize >> above, it was seen that : oneself may pushe the assumption which is the class / type of the lemma « goal1 » on top of the now-present goal, which is that oneself may *generalize the now-present goal by any deduction of the class / type* of the lemma « goal1 » . Yet another variation of this process is that instead of using some already globally-memorized lemma « goal1 », oneself may command « COQ » to create some new (internal) goal which is the class / type of « goal1 » and then to *generalize the old goal by any deduction of this class / type*. This process is named forward-generalize (for the « have » command) or backward-generalize (for the « suffices » command).

```
Lemma goal2 (x y1 y2 : nat) (x_gt1 : 1 < x)
   (odd_y1y2 : odd (y1 - y2)) : 2 < (y1 - y2) + x .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  2 < y1 - y2 + x
```

```
have : forall (y : nat), odd y -> 2 < y + x .
```

```
2 subgoals

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  forall y : nat, odd y -> 2 < y + x

subgoal 2 is:
 (forall y : nat, odd y -> 2 < y + x) -> 2 < y1 - y2 + x
```

```
by move => [ // | y' ] /= _ ; apply : ltn_addl x_gt1 .
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  (forall y : nat, odd y -> 2 < y + x) -> 2 < y1 - y2 + x
```

Now the old goal, which has been generalized :

```
by apply.
```

```
No more subgoals.
```

Alternatively, oneself may use the « suffices » command such to permute the printing-precedence of the old goal or the new goal. Additionnally, one may immediately solve the first-printed goal by postfixing the « have » or « suffices » command as follows.

```
Restart.
```

```
1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  2 < y1 - y2 + x
```

```
suffices : forall (y : nat), odd y -> 2 < y + x.
  by apply.

by move => [ // | y' ] /= _ ; apply : ltn_addl x_gt1 .
```

```
2 subgoals

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  (forall y : nat, odd y -> 2 < y + x) -> 2 < y1 - y2 + x

subgoal 2 is:
 forall y : nat, odd y -> 2 < y + x

1 subgoal

  x, y1, y2 : nat
  x_gt1 : 1 < x
  odd_y1y2 : odd (y1 - y2)
  ============================
  forall y : nat, odd y -> 2 < y + x

No more subgoals.
```

```
Qed.
```

5.3.8 weakening-generalize

Whenever oneself lacks to prevent copy-paste of some parts of the deduction programmer-script and also prevent copy-paste of some parts of the deduction computer-term, then this may originate from the presence of permutation-symmetry in the goal. For instance :

```
Lemma leq_max m n1 n2 : (m <= maxn n1 n2) = (m <= n1) || (m <= n2) .
```

Primo, memo that the conclusion of the goal it logically-equivalent to some change of itself where the variables « n1 » and « n2 » have been permuted, in other words « ((leq_max m) n1 n2 <-> (leq_max m) n2 n1) » holds « by rewrite maxnC orbC ». Secondo, the deduction starts by assuming each of the alternatives / cases of the destruction of the lemma « leq_total n2 n1 : (n2 <= n1) || (n1 <= n2) » . Finally, memo that all of the alternatives « (n2 <= n1) » or « (n1 <= n2) » are permutations of one single alternative, for instance « (n2 <= n1) » .

Therefore it shall be sufficient to deduce the conclusion under the assumption / weakener « (n2 <= n1) » .

```
suff : forall x y, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y).
```

```
2 subgoals

  m, n1, n2 : nat
  ============================
  (forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)) ->
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
 forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)
```

In other words, refering to the goal as « G » and to the weakener as « W », oneself has to deduce that the weakening / weakened goal « (W -> G) » is *sufficient* for the goal « G » as such « ((W -> G) -> G) » and then to deduce that the weakening « (W -> G) » indeed holds.

```
move : (leq_total n2 n1) => /orP (** this apply-in query the view hints database **) .
```

```
2 subgoals

  m, n1, n2 : nat
  ============================
  n2 <= n1 \/ n1 <= n2 ->
  (forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)) ->
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
 forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)
```

```
case => y_le_x => /(_ _ _ y_le_x) .
```

```
3 subgoals

  m, n1, n2 : nat
  y_le_x : n2 <= n1
  ============================
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2) ->
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
 (m <= maxn n2 n1) = (m <= n2) || (m <= n1) ->
 (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
subgoal 3 is:
 forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)
```

```
  by [].
by move => lem_perm; rewrite maxnC orbC.
```

```
2 subgoals

  m, n1, n2 : nat
  y_le_x : n1 <= n2
  ============================
  (m <= maxn n2 n1) = (m <= n2) || (m <= n1) ->
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
 forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)

1 subgoal

  m, n1, n2 : nat
  ============================
  forall x y : nat, y <= x -> (m <= maxn x y) = (m <= x) || (m <= y)
```

But wait ! Although the mathematical-sense did prevent the copy-paste of some parts of the deduction programmer-script and also did prevent the copy-paste of some parts of the deduction computer-term, memo that in the command « suff » above, oneself almost copy-pasted the class / type / statement of the lemma « leq_max » which shall be deduced.

Now to prevent this third variety of copy-paste, oneself may use the dedicated command « wlog » which does this weakening logic (wlog). This weakening logic, from the angle of view of permutations as-described-above, says that : one weakening (case) generates (all other permutation-case-weakenings of) the general goal (which is auto-permutative). The components of the weakening logic are : the weakener (« n2 <= n1 » , referred as « W ») , the occurrences of the variables in the weakened goal which shall be generalized (« n1 » and « n2 »), the sufficiency (« ((W -> G) -> G) ») of the (only-formally-)weakened goal, and the deduction of the weakened goal « (W -> G) » (where the weakener has been optionally pre-introduced by some name).

```
Restart.
```

```
1 subgoal

  m, n1, n2 : nat
  ============================
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

```
wlog le_n21 : n1 n2 / n2 <= n1 .
```

```
2 subgoals
```

```
  m, n1, n2 : nat
  ============================
  (forall n3 n4 : nat, n4 <= n3 -> (m <= maxn n3 n4) = (m <= n3) || (m <= n4)) ->
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)

subgoal 2 is:
 (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

```
by case/orP : (leq_total n2 n1) => y_le_x => /(_ _ _ y_le_x) ;
  last move => lem_perm; rewrite maxnC orbC.
```

```
1 subgoal

  m, n1, n2 : nat
  le_n21 : n2 <= n1
  ============================
  (m <= maxn n1 n2) = (m <= n1) || (m <= n2)
```

6 Deduction : data

Memo that the textual definition of classifications or classes such as « nat » mentions some (outer) meta-logical primitives such as alternatives « .. | .. » or inferences « .. -> .. » . And the textual definition of the data type / class « nat » is one of many ways of arranging-and-combining these meta-logical primitives. Oneself may attempt to simply write, for each (outer) meta-logical primitive, some textual definition which internalize / mimick / simulate this (outer) meta-logical primitive.

This section is limited for (data) types/classes, the next outline section 7 is for classifications.

6.1 inference

The pairing (or and …) data type internalizes the inference « .. -> .. » (outer) meta-logical primitive. This is communicated by the « Inductive » command which does some grammatical / inductive description / definition :

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

Primo as is common, the constructor function « pair » is the only alternative / case to (recursively) construct / build data in this class, as shown for instance in :

```
Check (@pair bool nat (false) (S (S O))).
```

```
(false, 2)
     : bool * nat
```

The « COQ » deduction / tactic command corresponding to the constructor function « pair » is « split » (or « constructor 1 ») ; therefore this deduction script generates the same deduction-term as the manually written deduction-term above :

```
Definition pair_false_2 : bool * nat .
```

```
  split.
    exact: false.
  exact: (S (S O)).
Defined.
```

```
1 subgoal

  ============================
  bool * nat

2 subgoals

  ============================
  bool

subgoal 2 is:
 nat

1 subgoal

  ============================
  nat

No more subgoals.

split.
 exact : false.

 exact : (S (S O)).

Defined.
pair_false_2 is defined
```

Secondo as is common, the only alternate constructor « pair » of the class « prod » computationally or logically fulfill / support this class, which is that it is sufficient to focus / touch on these (recursively) constructored data when holding this class, which is that any (random) data in the class may be such (recursively) destructed / eliminated / matched / filtered. This is described by one (grammatical) destructor / match function , as shown for instance in :

```
Definition fst (A B : Type) (p : prod A B) : A :=
  match p as p0 in prod _ _ return A with
    @pair _ _ a b => a
  end.
```

The « COQ » deduction / tactic command corresponding to this (grammatical) destructor / match function is « case » ; therefore this deduction script generates the same deduction-term as the manually written deduction-term above :

```
Definition fst (A B : Type) : prod A B -> A .
  case.
  move => a b ; exact : a.
Defined.
```

```
1 subgoal

  A : Type
  B : Type
  ============================
  A * B -> A

1 subgoal

  A : Type
  B : Type
  ============================
  A -> B -> A

No more subgoals.

case.
(move => a b; exact : a).

Defined.
fst is defined
```

6.2 classifying inference

The sigma (or ex …) data type internalizes the classifying inference « forall .. , .. » (outer) meta-logical primitive; memo that the *identifier « x » does not occur in the (inner-most) conclusion* of « existT » :

```
Inductive sigT (A : Type) (P : A -> Type) : Type :=
    existT : forall x : A, P x -> @sigT A P .
```

Primo, the « COQ » deduction / tactic command corresponding to the constructor function « existT » is « exists » (or « econstructor 1 » or « eexists ») :

```
Definition existsT_3_icons675inil : @sigT nat ilist .
  exists 3.
  exact: (icons 6 (icons 7 (icons 5 inil))).
  Show Proof.
Defined.
```

```
1 subgoal

  ============================
  {x : nat & ilist x}

1 subgoal

  ============================
  ilist 3

No more subgoals.

(existT ilist 3 (icons 6 (icons 7 (icons 5 inil))))

exists 3.
exact : (icons 6 (icons 7 (icons 5 inil))).

Defined.
existsT_3_icons675inil is defined
```

Secondo, the « COQ » deduction / tactic command corresponding to the (shared) (grammatical) destructor / match function is again « case » . The memo here is that the input may be re-arranged when this input does not affect the classification of the output.

6.3 alternative

The sum (or or …) data type internalizes the alternative « .. | .. » (outer) meta-logical primitive :

```
Inductive sum (A B : Type) : Type :=
    inl : A -> sum A B
  | inr : B -> sum A B .
```

Primo, the « COQ » deduction / tactic command corresponding to the constructor function « inl » is « left » (or « constructor 1 ») and the « COQ » deduction / tactic command corresponding to the constructor function « inr » is « right » (or « constructor 2 »). Secondo, the « COQ » deduction / tactic command corresponding to the (shared) (grammatical) destructor / match function is again « case » .

Additionally, the unit (or True …) data type or empty (or False …) data type correspondingly internalizes the 1-iterated or 0-iterated alternative « .. | .. » (outer) meta-logical primitive :

```
Inductive unit : Type :=  tt : unit .
Inductive empty : Type :=           .
```

Memo that zero constructor is sufficient to computationally or logically fulfill / support the class « empty » , which is that it is sufficient to touch none data when holding this class, which is that any (formal / grammatical) data in the class may be such destructed / eliminated / matched / filtered as none data. This is show for instance in :

```
Definition exfalso (A : Type) : empty -> A :=
  fun x : empty =>
    match x with end.
```

```
exfalso is defined
```

Elsewhere this deduced lemma « exfalso » also says that there is always some inference « empty -> A » ; therefore if oneself wants to say that « A <-> empty » (that « A » is empty), then it is sufficient to say that the inference type « A -> empty » is inhabitated. This type is commonly *shortened* as the definition « is_empty » (or « not »), instead of being described as some *grammatical / inductive definition / description* :

```
Definition is_empty (A : Type) := A -> empty .
```

6.4 recursion

Whenever oneself does some grammatical / inductive description / definition « idef1 » which recursively refers to itself « idef1 » in the input for some constructor function, then the recursion « fix .. := .. » grammatical (outer) meta-logical primitive is automatically internalized as some recursive elimination scheme / principle shortening-definition « idef1_rect » (and « idef1_ind » and « idef1_rec »). For instance, the « nat » numbers :

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

```
nat is defined
nat_rect is defined
nat_ind is defined
nat_rec is defined
```

```
Print nat_rect.
```

```
nat_rect =
fun (P : nat -> Type) (f : P 0) (f0 : forall n : nat, P n -> P (succn n)) =>
fix F (n : nat) : P n :=
  match n as n0 return (P n0) with
  | 0 => f
  | succn n0 => f0 n0 (F n0)
  end
     : forall P : nat -> Type,
       P 0 -> (forall n : nat, P n -> P (succn n)) -> forall n : nat, P n

Argument scopes are [function_scope _ function_scope nat_scope]
```

Memo that not all inductive descriptions are memorized by the « COQ » computer, for example when some constructor has the form « (cons1 : (idef1 -> type2) -> idef1) » ; this is such to prevent something as

```
Fail
Inductive empty_nonempty := Convert (empty : empty_nonempty -> False) .
```

```
The command has indeed failed with message:
Non strictly positive occurrence of "empty_nonempty" in
 "(empty_nonempty -> False) -> empty_nonempty".
```

```
Definition empty (nonempty : empty_nonempty) : False
  := let: Convert empty' := nonempty in empty' nonempty.

Check empty (Convert empty). (* : False *)
```

Primo as is common, the « COQ » deduction / tactic command corresponding to the constructor function « O » is « constructor 1 » (or « apply: O ») and the « COQ » deduction / tactic command corresponding to the constructor function « S » is « constructor 2 » (or « apply: S »).

Secondo as is common, the « COQ » deduction / tactic command corresponding to the shared (grammatical) destructor / match function is again « case ».

Tertio as is common, the « COQ » deduction / tactic command corresponding to the automatically-defined recursion scheme / principle « idef1_rect » is « elim » (or « apply: idef1_rect »). For instance :

```
Lemma addn0 : forall m : nat, m + 0 = m.
  elim.
```

```
1 subgoal

  ============================
  forall m : nat, m + 0 = m

2 subgoals

  ============================
  0 + 0 = 0

subgoal 2 is:
 forall n : nat, n + 0 = n -> succn n + 0 = succn n
```

and the value of the input function « P : nat -> Type » of the recursion scheme « nat_rect » (or « nat_ind ») which is automatically inferred (solved during higher-order unification …) by the « elim » (or « apply: nat_rect ») command is « (fun m' : nat => m' + 0 = m') » . Oneself may manually-synthesize « P » as follows (same as the « pattern » command) :

```
Undo.
move => m.
rewrite -[_ = _]/((fun m' => _) m) .
apply : nat_rect m (** or elim : m **) .
```

```
1 subgoal

  ============================
  forall m : nat, m + 0 = m

1 subgoal

  m : nat
  ============================
  m + 0 = m

1 subgoal

  m : nat
  ============================
  (fun m' : nat => m' + 0 = m') m

2 subgoals

  ============================
  0 + 0 = 0

subgoal 2 is:
 forall n : nat, n + 0 = n -> succn n + 0 = succn n
```

7 Deduction : classification

Memo that the textual definition of classifications or classes such as « nat » mentions some (outer) meta-logical primitives such as alternatives « .. | .. » or inferences « .. -> .. » . And the textual definition of the data type / class « nat » is one of many ways of arranging-and-combining these meta-logical primitives. Oneself may attempt to simply write, for each (outer) meta-logical primitive, some textual definition which internalize / mimick / simulate this (outer) meta-logical primitive.

This section is the continuation, for classifications, of the preceding outline section 6 for (data) types/classes.

7.1 meta-computation

The equality classifier internalizes (outer) meta-computation.

```
Print Coq.Init.Logic.eq.
```

```
Inductive eq (A : Type) (x : A) : A -> Prop :=  eq_refl : x = x

For eq: Argument A is implicit and maximally inserted
For eq_refl, when applied to no arguments:
  Arguments A, x are implicit and maximally inserted
For eq_refl, when applied to 1 argument:
  Argument A is implicit
For eq: Argument scopes are [type_scope _ _]
For eq_refl: Argument scopes are [type_scope _]
```

From the meta (outer) angle of view, « eq_refl » is some primitive deduction / proof term / value which says (asserts / deduces / proves) that the parameter « x » is "eq" ("equal") to any index-argument « y » which by-computation-is (by-convertibility-is) « x » ; and « eq_refl » is precisely-classified inside « @eq A x y » by any index-argument « y » which by-computation-is « x » . And the « COQ » deduction / tactic command corresponding to the constructor function « eq_refl » is « reflexivity » .

```
Check @Logic.eq_refl nat (3 + 2) : @Logic.eq nat (3 + 2) (S (S (1 + 2))).
```

```
erefl (3 + 2) : 3 + 2 = succn (succn (1 + 2))
     : 3 + 2 = succn (succn (1 + 2))
```

Moreover, deductions may be combinated in complex ways; which is that oneself may arrive at « @eq A x y » by more-complex combinaisons (by-logical-deduction) beyond precisely-one-primitive-deduction (by-computation) « eqrefl » .

From the meta (outer) angle of view, the description of "eq" above may be read as :

```
Inductive eq (A : Type) (x : A) : A -> Type :=
  eq_refl : forall ?y which by-computation-is x , @eq A x ?y .
```

Now the *elimination scheme* « eq_rect » (or « eq_ind »), which shorten the (grammatical) destructor/match function, has the sense of (congruent) rewriting / casting / transport.

```
eq_rect =
fun (A : Type) (x : A) (P : A -> Type) (f : P x) (y : A) (e : x = y) =>
match e in (_ = y0) return (P y0) with
| erefl _ _ => f
end
     : forall (A : Type) (x : A) (P : A -> Type),
       P x -> forall y : A, x = y -> P y

Argument A is implicit
Argument scopes are [type_scope _ function_scope _ _ _]
```

And as is common, the « COQ » deduction / tactic command corresponding to this (grammatical) destructor / match function is « case » . Moreover the alternative command « rewrite » is container of the « case »-command-for-equalities and enables more-complex pre-processing and post-processing. For example, one such pre-processing is doing more-easily the same generalization pre-processing which is enabled by the generalizing-case command « case : .. / .. » when *the index-argument of the equality classification is not some variable* :

```
Lemma lemma1337 (P : nat -> nat -> Type)
  (c : nat) (pcc : P c (c * c)) (Hc : c + c = 0)
  (Heq : forall (n : nat), c + c = n - n -> c = n * 0) n1 n2
  : P (n1 * 0) ((n2 * 0) * (n2 * 0)) .
```

```
case : ( n1 * _ ) / Heq.

Undo.
rewrite -[ n1 * _ ]Heq.

Undo.
rewrite -Heq.
```

```
2 subgoals

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  n1, n2 : nat
  ============================
  c + c = n1 - n1

subgoal 2 is:
 P c (n2 * 0 * (n2 * 0))

1 subgoal

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  Heq : forall n : nat, c + c = n - n -> c = n * 0
  n1, n2 : nat
  ============================
  P (n1 * 0) (n2 * 0 * (n2 * 0))

2 subgoals

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  Heq : forall n : nat, c + c = n - n -> c = n * 0
  n1, n2 : nat
  ============================
  P c (n2 * 0 * (n2 * 0))

subgoal 2 is:
 c + c = n1 - n1

1 subgoal

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  Heq : forall n : nat, c + c = n - n -> c = n * 0
  n1, n2 : nat
  ============================
  P (n1 * 0) (n2 * 0 * (n2 * 0))

2 subgoals

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  Heq : forall n : nat, c + c = n - n -> c = n * 0
  n1, n2 : nat
  ============================
  P c (n2 * 0 * (n2 * 0))

subgoal 2 is:
 c + c = n1 - n1
```

where, in the third command « rewrite -Heq. » , the subterms to rewrite are all the occurrences of the first match of the filter which has been inferred (here « (_ * 0) ») from the right-hand-side of the equality « Heq ».

And the rest of the modifiers for the « rewrite » command explains itself :

```
Fail rewrite -{2}Heq.
rewrite -{2}[ in X in P _ X ]Heq.
```

```
The command has indeed failed with message:
Ltac call to "rewrite (ssrrwargs) (ssrclauses)" failed.
Error: Only 1 < 2 occurence of the RHS
    (n1 * 0)
of Heq
1 subgoal

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  Heq : forall n : nat, c + c = n - n -> c = n * 0
  n1, n2 : nat
  ============================
  P (n1 * 0) (n2 * 0 * (n2 * 0))

2 subgoals

  P : nat -> nat -> Type
  c : nat
  pcc : P c (c * c)
  Hc : c + c = 0
  Heq : forall n : nat, c + c = n - n -> c = n * 0
  n1, n2 : nat
  ============================
  P (n1 * 0) (n2 * 0 * c)

subgoal 2 is:
 c + c = n2 - n2
```

where any « [in ..] » modifier such as « [in X in P _ X] » is named some imprecise-contextual filter, and any « { .. } » modifier such as « {2} » is named some occurence modifier, further more :

```
rewrite -{2}[ (_ * 0) in X in P _ X ]Heq
  ?Hc ?subnn // (** 1 branch **)
  -!Heq // (** trunk **)
  Hc subnn // (** 2 branches **) .
```

```
No more subgoals.
```

where any « [(..) in ..] » modifier such as « [(_ * 0) in X in P _ X] » is named some more-precise-contextual filter , where the lemma « subnn : forall n : nat, n - n = 0 » , and further more : where the sequencing of the « ? » modifier and the « // » modifier is sometimes used

- to early solve / close any generated branching / side conditions of the rewriting-equation, and then continue with the trunk goal, or
- to early solve / close the trunk goal and then continue with the branching / side conditions.

Elsewhere, because shortening-definitions (such as « leq ») may be unfolded by-computation (*delta rule …*), and because by-computation (« eq_refl ») is some (primitive) deduction of equality, therefore the « rewrite » command may be inputed the name of some shortening-definition which shall be unfold-rewritten / unfolded :

```
Lemma leq_mul2l m n1 n2 :
  (m * n1 <= m * n2) = (eqn m 0) || (leq n1 n2).
```

```
1 subgoal

  m, n1, n2 : nat
  ============================
  (m * n1 <= m * n2) = eqn m 0 || (n1 <= n2)
```

```
rewrite [in LHS]/leq.
```

```
1 subgoal

  m, n1, n2 : nat
  ============================
  (m * n1 - m * n2 == 0) = eqn m 0 || (n1 <= n2)
```

where the *imprecise contextual filter* « [in LHS] » is short notation for « [in X in X = _] » , and the « == » notation is some more convoluted way ("canonical structures") of saying « eqn » …

More generally, the « rewrite » command (and the corresponding « change » command …) may be used to change-rewrite / change / replace some subterm of the goal by another term whenever these two terms are equal-by-computation :

```
rewrite -[n1]/(addn 0 n1).
```

```
1 subgoal
```

```
  m, n1, n2 : nat
  ============================
  (m * (0 + n1) - m * n2 == 0) = eqn m 0 || (0 + n1 <= n2)
```

Another angle of view of the change-rewrite command is that it fold-rewrite / fold the *head constant* of the destination term (filter), where the inferred filter which shall be matched in the goal is the filter which is obtained after unfolding this head constant of the destination filter.

```
rewrite -/(leq _ _).
```

```
1 subgoal

  m, n1, n2 : nat
  ============================
  (m * (0 + n1) <= m * n2) = eqn m 0 || (0 + n1 <= n2)
```

7.2 boolean-computational reflection

The reflect classifier internalizes the decidability of decidable types (predicates).

```
Inductive reflect (P : Type) : bool -> Type :=
| ReflectT (p : P) : reflect P true
| ReflectF (np : P -> False) : reflect P false.
```

```
reflect is defined
reflect_rect is defined
reflect_ind is defined
reflect_rec is defined
```

This reflect classifier « reflect P b » uses some binary / boolean (inner) index-argument « b » to classify the presence or absence of data in some (outer) parameter type « P ». The *elimination scheme* « reflect_rect » contains some sense of *automatic substitution* of the occurrences of the index-argument in the goal; this is comparable to the equality classifier.

Now instead of *rewrite lemmas* for the equality classifier, onself has *reflection lemmas* for the reflect classifier. For instance, which also shows that the inputs « P » and « b » in any textual description « reflect P b » may be memorized as *logical predicate* and *boolean predicate* :

```
About andP.
```

```
andP : forall b1 b2 : bool, reflect (b1 /\ b2) (b1 && b2)

Arguments b1, b2 are implicit and maximally inserted
Argument scopes are [bool_scope bool_scope]
andP is opaque
Expands to: Constant mathcomp.ssreflect.ssrbool.andP
```

```
Lemma example37 a b : a && b ==> (a == b).

  case : andP .
```

```
1 subgoal

  a, b : bool
  ============================
  a && b ==> (a == b)

2 subgoals

  a, b : bool
  ============================
  a /\ b -> true ==> (a == b)

subgoal 2 is:
 ~ (a /\ b) -> false ==> (a == b)
```

Memo that the equality classifier has some alternative command « rewrite » which is container of the generalizing-case command « case : .. / .. » for equalities, but there is no such command for the « reflect » classifier and the generalizing-case command « case : .. / .. » shall be queried directly.

```
Lemma example13 a b : (a || ~~ a) && (a && b ==> (a == b)).

  case E: (a && _) / andP.
```

```
1 subgoal

  a, b : bool
  ============================
  (a || ~~ a) && (a && b ==> (a == b))

2 subgoals

  a, b : bool
  _p_ : a /\ b
  E : a && b = true
  ============================
  (a || ~~ a) && (true ==> (a == b))

subgoal 2 is:
 (a || ~~ a) && (false ==> (a == b))
```

And oneself shall use these lemmas if oneself lacks to deduce new reflection lemmas :

```
About idP.
About iffP.
```

```
idP : forall b1 : bool, reflect b1 b1

Argument b1 is implicit and maximally inserted
Argument scope is [bool_scope]
idP is opaque
Expands to: Constant mathcomp.ssreflect.ssrbool.idP

iffP :
forall (P Q : Prop) (b : bool),
reflect P b -> (P -> Q) -> (Q -> P) -> reflect Q b
```

```
Arguments P, Q, b are implicit
Argument scopes are [type_scope type_scope bool_scope _ function_scope
  function_scope]
iffP is opaque
Expands to: Constant mathcomp.ssreflect.ssrbool.iffP
```

7.3 boolean simultaneous-substitution

The compare_nat classifier internalizes the totality of the ordering on numbers.

```
Inductive compare_nat (m n : nat) : bool -> bool -> bool -> Set :=
    CompareNatLt : m < n -> compare_nat m n true false false
  | CompareNatGt : n < m -> compare_nat m n false true false
  | CompareNatEq : m = n -> compare_nat m n false false true .
```

```
compare_nat is defined
compare_nat_rect is defined
compare_nat_ind is defined
compare_nat_rec is defined
```

And with this comparaison lemma « ltngtP » , it may be used to do boolean simultaneous-substitution of all the occurrences of « (m < n) » and « (n < m) » and « (m == n) » in the goal :

```
Check ltngtP : forall m n, compare_nat m n (m < n) (n < m) (m == n) .
```

Another comparaison classifier is « leq_xor_gtn » , which comes with some comparaison lemma « leqP » .

7.4 recursion

For *classifiers*, as for *classes*, the recursion « fix .. := .. » grammatical (outer) meta-logical primitive is automatically internalized as some recursive elimination scheme shortening-definition. For instance, the « ilist » indexed lists :

```
Inductive ilist : nat -> Type :=
    inil : ilist 0
  | icons : nat -> forall m : nat, ilist m -> ilist (S m).
```

```
ilist is defined
ilist_rect is defined
ilist_ind is defined
ilist_rec is defined
```

```
ilist_rect =
fun (P : forall n : nat, ilist n -> Type) (f : P 0 inil)
  (f0 : forall (n m : nat) (i : ilist m), P m i -> P (succn m) (icons n i))
=>
fix F (n : nat) (i : ilist n) {struct i} : P n i :=
  match i as i0 in (ilist n0) return (P n0 i0) with
  | inil => f
  | @icons n0 m i0 => f0 n0 m i0 (F m i0)
  end
     : forall P : forall n : nat, ilist n -> Type,
       P 0 inil ->
       (forall (n m : nat) (i : ilist m), P m i -> P (succn m) (icons n i)) ->
       forall (n : nat) (i : ilist n), P n i

Arguments P, n are implicit
Argument scopes are [function_scope _ function_scope nat_scope _]
```

In contrast to classes, for classifiers, the « COQ » deduction / tactic command corresponding to the shared (grammatical) destructor / match function is some more complex generalizing-case command « case : .. / .. » because *the index-argument of the classifier-which-shall-be-destructed may not be some variable-identifier*. Memo that the « rewrite » command is some container of this same generalizing-case command for the *equality classifier*.

For instance, this function « ihead_ibehead » input from some indexed list whose index (which simulates the *real size* number of items in the list) is some positive number and output the pair where the first component is the head item of the list and the second component is the rest of the list :

```
Definition ihead_ibehead (m : nat) (l : ilist (S m)) : nat * (ilist m).

case Heq : (S _) / l => [ | j m' l' ] .
```

```
1 subgoal

  m : nat
  l : ilist (succn m)
  ============================
  nat * ilist m

2 subgoals

  m : nat
  Heq : succn m = 0
  ============================
  nat * ilist m

subgoal 2 is:
 nat * ilist m
```

Oneself shall exfalso this first goal :

```
suff : False by exact: (fun f : False => match f with end).
```

```
2 subgoals

  m : nat
  Heq : succn m = 0
  ============================
  False

subgoal 2 is:
 nat * ilist m
```

then the equation « Heq : m.+1 = 0 » shall enable rewrite / transport / cast of data from any origin class « A » corresponding to « m.+1 » into any other destination class « B » corresponding « 0 » , because « m.+1 » is discriminable / distinguishable from « 0 » such that oneself is able to *program such classification* of « A » by « m.+1 » and « B » by « 0 ». Here the destination class « B » is « False » and the origin class is « True » with its data « I ».

```
rewrite -[False]/((fun n : nat => match n with
                       | _.+1 => True
                       | 0 => False
                       end) 0).
```

```
2 subgoals

  m : nat
  Heq : succn m = 0
  ============================
  (fun n : nat => match n with
                  | 0 => False
                  | succn _ => True
                  end) 0

subgoal 2 is:
 nat * ilist m
```

```
rewrite -[0]Heq.
exact : I.
```

```
2 subgoals

  m : nat
  Heq : succn m = 0
  ============================
  True

subgoal 2 is:
 nat * ilist m

1 subgoal

  m, j, m' : nat
  l' : ilist m'
  Heq : succn m = succn m'
  ============================
  nat * ilist m
```

Alternatively oneself may use the « by [] » composite-command to automatically deduce this *discrimination lemma* :

```
Undo 4.
by [].
```

```
2 subgoals

  m : nat
  Heq : succn m = 0
  ============================
  nat * ilist m
```

```
subgoal 2 is:
 nat * ilist m

1 subgoal

  m, j, m' : nat
  l' : ilist m'
  Heq : succn m = succn m'
  ============================
  nat * ilist m
```

Now the second goal starts by deducing some new equation from the old equation « Heq » by beholding the injectivity of constructor functions (here the successor constructor function has some *named cancelling function* which is the predecessor function « _ .-1 »)

```
have Heq_injective : m.+1.-1 = m'.+1.-1 by rewrite Heq.
```

```
1 subgoal

  m, j, m' : nat
  l' : ilist m'
  Heq : succn m = succn m'
  Heq_injective : predn (succn m) = predn (succn m')
  ============================
  nat * ilist m
```

Alternatively oneself may use the « case » command to automatically deduce this injectivity lemma :

```
Undo. case : Heq => Heq_injective.

rewrite [m]Heq_injective.
exact: (j , l').
Defined.
```

```
1 subgoal

  m, j, m' : nat
  l' : ilist m'
  Heq : succn m = succn m'
  ============================
  nat * ilist m

1 subgoal

  m, j, m' : nat
  l' : ilist m'
  Heq_injective : m = m'
  ============================
  nat * ilist m

1 subgoal

  m, j, m' : nat
  l' : ilist m'
  Heq_injective : m = m'
  ============================
  nat * ilist m'

No more subgoals.

case  Heq: (S _) / l => [|j m' l'].
 by [  ].

 case : Heq => Heq_injective.
 rewrite [m]Heq_injective.
 exact : (j, l').

Defined.
ihead_ibehead is defined
```

In contrast to classes, for classifiers, the « COQ » deduction / tactic command corresponding to the recursive elimination scheme is some more complex generalizing-elimination command « elim : .. / .. » because *the index-argument of the classifier-which-shall-be-destructed may not be some variable-identifier*.

Finally, some lemmas may be solved more-sensibly if the deduction uses some alternative recursion scheme which is manually-defined by the programmer. This is done by using the « elim / scheme1 » command. For instance, oneself may view « addn » as some classification which is its *graph* and deduce the manually-defined recursion schema :

```
Lemma addn_classifier (m : nat) (P : nat -> nat -> Prop) (H: P 0 m)
 (H0 : forall  p : nat, P p (addn p m) -> P p.+1 (addn p m).+1)
 : forall n : nat, P n (addn n m) .

elim; [apply: H | apply: H0]. Qed.
```

Then the deduction of this associativity lemma is smoother because some extra simplification-computation step is avoided …

```
Lemma exF x y z: addn z (addn y x) = addn (addn z y) x.
```

```
elim/addn_classifier : z / (addn z _) .
```

```
2 subgoals

  x, y : nat
  ============================
  y + x = 0 + y + x

subgoal 2 is:
 forall p : nat,
 p + (y + x) = p + y + x -> succn (p + (y + x)) = succn p + y + x
```

```
  by [].
by move => p -> .
```

8 Review of some long deductions

8.1 Accumulating division

Accumulating (euclidian) division is

- iterating subtraction on some input data until the remaining input data (the remainder) is too small for another substraction, and simultaneously
- memorizing the number of iterations (the quotient) into some extra accumulator-memory.

```
Definition edivn_rec d :=
  fix loop m q := if m - d is m'.+1 then loop m' q.+1 else (q, m).

Definition edivn m d := if d > 0 then edivn_rec d.-1 m 0 else (0, m).
```

Memo that And the « COQ » computer is very good at detecting that the accumulator « m » is degrading for the « fix » command, although the term « m' » is not some *immediate-subterm* of « m ». Elsewhere the « edivn » program handles the case of some zero divisor, producing the dummy pair « (0,m) » for the quotient and the remainder correspondingly.

It is sometimes useful to describe and deduce unfolding equations like this one :

```
Lemma edivn_recE d m q :
  edivn_rec d m q = if m - d is m'.+1 then edivn_rec d m' q.+1 else (q,m).

 by case: m. Qed.
```

Indeed the simplification tactic « /= » (or « simpl ») may unfold excessively than whatever is wanted. Rewriting with such equations enables finer communcation of how many unfold steps shall be performed.

Now oneself shall deduce that the « edivn » *program satisfies some given property / specification* :

```
Lemma edivnP m d (ed := edivn m d) :
  ((d > 0) ==> (ed.2 < d)) && (m == ed.1 * d + ed.2).
```

The type of this lemma uses some let-in / behold / locally-defined parameter / hypothesis « (.. := ..) » which translates as some « let .. := .. in .. » term.

Oneself shall deduce this lemma by lessorequal-generalizing induction (strong induction) because the recursive query of the function « edivn_recE » is made at some deeper nonimmediate-subterm of the accumulator.

Primo, the case of « d » being zero is solved, where oneself uses the *revert-then-intro contextualization* « .. in .. » of the command « case ».

```
case: d => [|d /=] in ed *; first by rewrite eqxx.
```

```
1 subgoal

  m, d : nat
  ed := edivn m (succn d) : nat * nat
  ============================
  (ed.2 < succn d) && (m == ed.1 * succn d + ed.2)
```

Then the next commands shall do some pre-processing before the induction by unfolding the definition of edivn (to expose the initial value of the accumulators of « edivn_rec ») and makes the *invariant of the division loop* explicit replacing « m » by « (0 * d.+1 + m) » .

```
rewrite -[edivn m d.+1]/(edivn_rec d m 0) in ed *.
rewrite -[m]/(0 * d.+1 + m).
```

```
1 subgoal

  m, d : nat
  ed := edivn_rec d m 0 : nat * nat
  ============================
  (ed.2 < succn d) && (m == ed.1 * succn d + ed.2)

1 subgoal

  m, d : nat
  ed := edivn_rec d m 0 : nat * nat
  ============================
  (ed.2 < succn d) && (0 * succn d + m == ed.1 * succn d + ed.2)
```

Then the next command does some behold / local-definition generalization pre-processing by prefixing the « ed » beholding with the « @ » modifier. Then the same composite-command does *lessorequal-generalization* pre-processing, then some *initial-accumulator-generalization* pre-processing, then query the *recursive elimination scheme* for « nat » numbers. Finally the same composite-command does some post-processing such as introducing some identifier-names and immediately-solving some goals.

```
elim: m {-2}m 0 (leqnn m) @ed => [[]//=|n IHn [//=|m]] q le_mn.
```

```
1 subgoal

  d, n : nat
  IHn : forall m n0 : nat,
        m <= n ->
        let ed := edivn_rec d m n0 in
        (ed.2 < succn d) && (n0 * succn d + m == ed.1 * succn d + ed.2)
  m, q : nat
  le_mn : m < succn n
  ============================
  let ed := edivn_rec d (succn m) q in
  (ed.2 < succn d) && (q * succn d + succn m == ed.1 * succn d + ed.2)
```

The following lemmas shall be used in the next command :

```
About subn_if_gt.
About negbT.
```

```
subn_if_gt :
forall (T : Type) (m n : nat) (F : nat -> T) (E : T),
match succn m - n with
| 0 => E
| succn m' => F m'
end = (if n <= m then F (m - n) else E)

Argument T is implicit
Argument scopes are [type_scope nat_scope nat_scope function_scope _]
subn_if_gt is opaque
Expands to: Constant mathcomp.ssreflect.ssrnat.subn_if_gt
```

```
negbT : forall b : bool, b = false -> ~~ b

Argument b is implicit
Argument scopes are [bool_scope _]
negbT is opaque
Expands to: Constant mathcomp.ssreflect.ssrbool.negbT
```

This next command unfolds the recursive function along the lemma « edivn_recE » and uses the lemma « subn_if_gt » to push the subtraction into the branches of the if statement. Then the following command pre-process the condition « (d <= m) » of the if-then-else statement by doing some *equational-generalize* of this condition, then destructs this condition to expose the alternative cases, and finally post-process the memorizing equation through branching and introduction filters which use the *apply-in view* « /negbT ». Memo that the shorter equational-generalize command « case E : (d <= m) » may also be used instead of this long manual equational-generalize.

```
rewrite edivn_recE subn_if_gt;
case : {-1}(d <= m) (erefl (d <= m)) => [le_dm | /negbT lt_md]; last first.
```

```
2 subgoals

  d, n : nat
  IHn : forall m n0 : nat,
        m <= n ->
        let ed := edivn_rec d m n0 in
        (ed.2 < succn d) && (n0 * succn d + m == ed.1 * succn d + ed.2)
  m, q : nat
  le_mn : m < succn n
  lt_md : ~~ (d <= m)
  ============================
  let ed := (q, succn m) in
  (ed.2 < succn d) && (q * succn d + succn m == ed.1 * succn d + ed.2)

subgoal 2 is:
 let ed := edivn_rec d (m - d) (succn q) in
 (ed.2 < succn d) && (q * succn d + succn m == ed.1 * succn d + ed.2)
```

The else branch corresponds to the non recursive case of the division algorithm and is immediately-solved by this command :

```
by rewrite /= ltnS ltnNge lt_md eqxx.
```

```
1 subgoal

  d, n : nat
  IHn : forall m n0 : nat,
        m <= n ->
        let ed := edivn_rec d m n0 in
        (ed.2 < succn d) && (n0 * succn d + m == ed.1 * succn d + ed.2)
  m, q : nat
  le_mn : m < succn n
  le_dm : (d <= m) = true
  ============================
  let ed := edivn_rec d (m - d) (succn q) in
  (ed.2 < succn d) && (q * succn d + succn m == ed.1 * succn d + ed.2)
```

The recursive query is done on « (m-d) », hence the lack for *lessorequal-generalizing* *induction* (*strong induction*). The premise for the induction hypothesis « (m - d <= n) » is deduced in some separate foward step via the *forward-generalize* « have » command. This same command does some post-processing *specialization* view introduction filter onto this intermediate lemma which is now some extra assumption of the old goal.

```
have /(IHn _ q.+1) : m - d <= n by rewrite (leq_trans (leq_subr d m)).
```

```
1 subgoal

  d, n : nat
  IHn : forall m n0 : nat,
        m <= n ->
        let ed := edivn_rec d m n0 in
        (ed.2 < succn d) && (n0 * succn d + m == ed.1 * succn d + ed.2)
  m, q : nat
  le_mn : m < succn n
  le_dm : (d <= m) = true
  ============================
  (let ed := edivn_rec d (m - d) (succn q) in
   (ed.2 < succn d) && (succn q * succn d + (m - d) == ed.1 * succn d + ed.2)) ->
  let ed := edivn_rec d (m - d) (succn q) in
  (ed.2 < succn d) && (q * succn d + succn m == ed.1 * succn d + ed.2)
```

This final command deduces that some cancellation of numbers indeed does occur :

```
by rewrite /= mulSnr -addnA -subSS subnKC.
Qed.
```

```
No more subgoals.

(<ssreflect_plugin::ssrtclseq@0> case : d => [|d /=] in   ed * ; first  by
 rewrite eqxx).
rewrite -[edivn m d.+1]/(edivn_rec d m 0) in   ed *.
rewrite -[m]/(0 * d.+1 + m).
elim : m {-2}m 0 (leqnn m) @ed => [[] //=|n IHn [//=|m]] q le_mn.
(<ssreflect_plugin::ssrtclseq@0>
 rewrite edivn_recE subn_if_gt; case : {-1}(d <= m)
  (erefl (d <= m)) => [le_dm|/negbT lt_md] ; last  first).
 by rewrite /= ltnS ltnNge lt_md eqxx.

 have /(IHn _ q.+1): m - d <= nby rewrite (leq_trans (leq_subr d m)).
 by rewrite /= mulSnr -addnA -subSS subnKC.

Qed.
edivnP is defined
```

8.2 Review 1337777 Solution Programme

The « SOLUTION PROGRAMME » is some continuation of the « DOSEN PROGRAMME » [1] and the « COQ MATH-COMP PROGRAMME » [2] , [3].

The « SOLUTION PROGRAMME » has discovered [4] , [5] that the attempt to deduce associative coherence by Maclance is in fact not the reality, because this famous pentagone is in fact some recursive square.

Moreover the « SOLUTION PROGRAMME » has discovered [6] , [7] that the categories only-named by the homologist Maclane are in fact the natural polymorphism of the logic of Gentzen, this enables some programming of congruent resolution by cut-elimination [8] which will serve as specification (reflection) technique to semi-decide the questions of coherence, in comparasion from the ssreflect-style.

Furthermore the « SOLUTION PROGRAMME » has discovered [9] , [10] that the Galois-action for the resolution-modulo, is in fact some instance of polymorph functors.

And the « SOLUTION PROGRAMME » has discovered [11] , [12] how to program polymorph coparametrism functors ("comonad").

Additionnally, the « SOLUTION PROGRAMME » has discovered information-technology [13] , [14] , [15] based on the EMACS org-mode logiciel which enables communication of timed-synchronized geolocated simultaneously-edited multi-authors searchable text, and therefore communication of textual COQ math programming, and which enables webcitations / reviews.

Whatever is discovered, its format, its communication is simultaneously some predictable-time (1337) logical resolution and some random-moment (777) dia-para-logical resolution. This text is one possible review of some of the 1337777 Solution Programme, which is continuing forever-in-time …

9 ConfusPlay

9.1 ConfusPlay 1

Correct this « all_words » program which input from some length « (n : nat) » and some sequence of symbols alphabet « (alphabet : seq T) » and shall generate some list containing any word-of-size-n, which signifies : some list of list-of-size-n.

```
Eval compute in all_words 2 [:: 1; 2; 3].
```

```
= [:: [:: 1; 1]; [:: 1; 2]; [:: 1; 3];
[:: 2; 1]; [:: 2; 2]; [:: 2; 3];
[:: 3; 1]; [:: 3; 2]; [:: 3; 3]]
```

```
Definition all_words (n : nat) (T : Type) (alphabet : seq T) :=
  let prepend x wl := [seq x :: w | w <- wl] in
  let extend wl := flatten [seq prepend x wl | x <- alphabet] in
    iter n extend [::].
```

9.2 ConfusPlay 2, refer ConfusPlay 1

Complete this « size_all_words » lemma by filling-in the two blank spaces marked by « (** ???1 **) » and « (** ???2 **) » . Then separate this lemma in two distincts lemmas « size_extend » and « size_all_words » such that this nested-(double)-induction-in-single-lemma is erased.

```
Lemma size_all_words n T (alphabet : seq T) :
  size (all_words n alphabet) = size alphabet ^ n.

elim: n => [|n IHn]; first by rewrite expn0.
rewrite expnS -{}(** ???1 **) [in LHS]/all_words iterS -/(all_words _ _).

elim: alphabet (all_words _ _) => //= w ws IHws aw.
by rewrite size_cat (** ???2 **) size_map mulSn.
Qed.
```

10 Reviews

[1] « DOSEN » http://www.mi.sanu.ac.rs/~kosta

[2] « COQ » https://coq.inria.fr

[3] « 1337777.OOO » https://1337777.github.io/init.html

[4] « 1337777.OOO » https://github.com/1337777/dosen/blob/master/coherence2.v

[5] « 1337777.OOO » https://github.com/1337777/dosen/blob/master/coherence.v

[6] « 1337777.OOO » https://github.com/1337777/borceux/blob/master/borceuxSolution2.v

[7] « 1337777.OOO » https://github.com/1337777/borceux/blob/master/chic05.pdf

[8] « 1337777.OOO » https://github.com/1337777/dosen/blob/master/dosenSolution3.v

[9] « 1337777.OOO » https://github.com/1337777/aigner/blob/master/aignerSolution.v

[10] « 1337777.OOO » https://github.com/1337777/aigner/blob/master/ocic04-where-is-combinatorics.pdf

[11] « 1337777.OOO » https://github.com/1337777/laozi/blob/master/ocic03-what-is-normal.djvu

[12] « 1337777.OOO » https://github.com/1337777/laozi/blob/master/laoziSolution2.v

[13] « 1337777.OOO » https://github.com/1337777/OOO1337777/blob/master/makegit.sh.org

[14] « 1337777.OOO » http://1337777.link/ooo/guJAH-jSeQcTEST/2016/1/5/11/9/11/1

[15] « 1337777.OOO » https://github.com/1337777/upo/blob/master/editableTree.urp