Number Notation dans Coq (démonstration)

Pierre Roux

Number Notation dans Coq (démonstration)

Pierre Roux[1]

ONERA/DTIS, Université de Toulouse, F-31055 Toulouse - France pierre.roux@onera.fr

Résumé

L'assistant de preuve Coq dispose d'une fonctionnalité permettant à l'utilisateur de définir ses propres notations avec une grande souplesse. Dans ce cadre, les constantes numériques étaient initialement interprétées et affichées par du code dédié dans des plugins OCaml. Depuis Coq 8.9 (janvier 2019), la commande Numeral Notation (aujourd'hui renommée Number Notation) permet d'implémenter ces parsers et printers de constantes numériques directement dans Coq.

Suite entre autre à l'introduction des flottants primitifs, cette commande a depuis connue un certain nombre d'extensions qui seront présentées dans cette démonstration : valeurs à virgules ou exposant, valeurs hexadécimales, interprétation vers des types inductifs paramétrés ou non inductifs.

1 La commande `Number Notation`

Dans Coq [1], depuis janvier 2019 et sa version 8.9, la commande `Number Notation` permet de déclarer directement dans Coq des parsers et printers pour les nombres entiers. Cela les rend plus facile à écrire, modifier et maintenir mais permet également de prouver leur bon comportement, assurant qu'ils n'introduisent pas d'incohérence de Pollack [3].

Ces entiers suivent l'expression régulière `-? [0-9]+` et sont, par défaut, interprétés comme des entiers naturels de type `nat`, comme on peut le constater en demandant à Coq `Check 42`. Cela est obtenu grâce au code suivant dans le prélude de la librairie standard de Coq (fichier `theories/Init/Prelude.v`) :

```
Number Notation nat Nat.of_num_uint Nat.to_num_uint (abstract after 5001)
  : nat_scope.
```

avec un parser `Nat.of_num_uint` de type `Number.uint -> nat` et un printer `Nat.to_num_uint` de type `nat -> Number.uint`. Le type `Number.uint` encapsulant un type `Decimal.uint` représentant les nombres comme listes de chiffres

```
Inductive uint : Set :=
  | Nil : Decimal.uint
  | D0 : Decimal.uint -> Decimal.uint
  | D1 : Decimal.uint -> Decimal.uint
  (* ... *)
  | D9 : Decimal.uint -> Decimal.uint
```

Les parsers/printers peuvent être des fonctions partielles

```
Definition bool_of_uint : Number.uint -> option bool := (* ... *)
Definition uint_of_bool : bool -> Number.uint := (* ... *)
Number Notation bool bool_of_uint uint_of_bool : bool_scope.
```

et on peut parser des valeurs signées en remplaçant `Number.uint` par `Number.int`.

2 Nombres décimaux

Depuis Coq 8.10 (octobre 2019), on dispose de nombres avec virgule et exposant par exemple pour les rationnels de la librairie standard

```
Require Import QArith. Local Open Scope Q_scope. Check 1.23e-2.
```

suivant l'expression régulière `-? [0-9]+ ( . [0-9]+ )? ( [eE] [+-]? [0-9]+ )?`. Le type `Decimal.decimal` est alors utilisé, similairement à `Decimal.uint` ci dessus.

3 Nombres hexadécimaux

Depuis Coq 8.12 (juillet 2020), on dispose également d'une syntaxe hexadécimale (par exemple `Check 0x2a.`) suivant l'expression `-? ( 0x | 0X ) [0-9a-fA-F]+ ( . [0-9a-fA-F]+ )? ( [pP] [+-]? [0-9]+ )?` autorisant les virgules et exposant (binaire), utilisés par exemple pour afficher exactement les nombres flottants.

4 Commentaires

Des `_` peuvent être utilisés comme commentaires pour rendre des grandes constantes plus lisibles : `Check 1_000_000.`

5 Types non inductifs

Pour pouvoir facilement appeler les fonctions de print quand un terme est affiché et pouvoir faire des match, les types parsés doivent être des inductifs, éventuellement paramétrés.

Certains types numériques ne sont pas inductifs, par exemple les réels de la librairie standard [2]. La commande `Number Notation` permet de parser vers ces types, en utilisant un type inductif ad hoc et en établissant une correspondance entre cet inductif et des constantes du type concerné. Par exemple pour les réels (c.f., fichier `theories/Reals/Rdefinitions.v`), on déclare le type `IR` puis on décrit une association entre les constructeurs de ce type et les constantes de type `R` qui nous intéressent

```
Require Import Reals.
Print IR.
Number Notation R of_number to_number (via IR
  mapping [IZR => IRZ, Q2R => IRQ, Rmult => IRmult, Rdiv => IRdiv,
           Z.pow_pos => IZpow_pos, Z0 => IZ0, Zpos => IZpos, Zneg => IZneg])
  : R_scope.
```

6 Conclusion

Aujourd'hui dans Coq master, grâce à ces mécanismes, tous les anciens parsers/printers numériques sous forme de plugins OCaml ont pu être remplacés par des implémentations en Coq. Cela les rend plus facile à écrire, modifier et maintenir mais permet également de prouver leur bon comportement, assurant qu'ils n'introduisent pas d'incohérence de Pollack [3]. On peut trouver de telles preuves dans les fichiers `theories/Numbers/Decimal*.v` de la librairie standard par exemple.

Références

[1] The Coq development team. *The Coq proof assistant reference manual*, 2021. Version 8.13.

[2] Micaela Mayero. *Formalisation et automatisation de preuves en analyses réelle et numérique*. PhD thesis, Université Paris VI, décembre 2001.

[3] Freek Wiedijk. Pollack-inconsistency. *Electronic Notes in Theoretical Computer Science*, 285 :85–100, 2012. Proceedings of the 9th International Workshop On User Interfaces for Theorem Provers (UITP10).

A Fichier Coq de la démonstration

```
(* Démonstration réalisée avec Coq 8.13 *)

(* Jusque récemment (Coq 8.10), le parseur de Coq lisait
   des nombres entiers : -? [0-9]+

   Par défaut interprété comme des nat *)
Check 42.

(* grâce au code suivant dans le prélude de la librairie standard
   (fichier theories/Init/Prelude.v) *)
Number Notation nat Nat.of_num_uint Nat.to_num_uint (abstract after 5001)
  : nat_scope.

(* avec un parseur de type Number.uint -> nat *)
Check Nat.of_num_uint.

(* et un printer de type nat -> Number.uint *)
Check Nat.to_num_uint.

(* avec Number.uint *)
Print Number.uint.
(* et Decimal.uint des listes de chiffres de 0 à 9 *)
Print Decimal.uint.

(* le [abstract after 5001] évite des débordements de pile
   en n'évaluant pas le parseur pour les grands nombres *)
Check 12000.

(* et [: nat_scope] place la notation dans le scope [nat_scope]
   (ouvert par défaut) *)

(* On peut parser des valeurs signées
   en remplaçant Number.uint par Number.int *)
Print Decimal.int.

(* Les parseur/printer peuvent être des fonctions partielles : *)

Definition bool_of_uint : Number.uint -> option bool :=
  fun u =>
    match u with
    | Number.UIntDecimal (Decimal.D0 Decimal.Nil) => Some false
    | Number.UIntDecimal (Decimal.D1 Decimal.Nil) => Some true
    | _ => None
    end.

Definition uint_of_bool : bool -> Number.uint :=
```

```
  fun b =>
    if b then Number.UIntDecimal (Decimal.D1 Decimal.Nil)
    else Number.UIntDecimal (Decimal.D0 Decimal.Nil).

Number Notation bool bool_of_uint uint_of_bool : bool_scope.

Check 0%bool.
Check 1%bool.
Fail Check 2%bool.

(*********************)
(* Nombres décimaux *)
(*********************)

(* Depuis Coq 8.10, on dispose de nombres avec virgule et exposant :
   -? [0-9]+ ( . [0-9]+ )? ( [eE] [+-]? [0-9]+ )? *)

(* Par exemple pour les rationnels de la lib standard *)
Require Import QArith.
Local Open Scope Q_scope.
Check 1.23e-2.

Goal 1.23e-2 = 0.0123.
Proof. reflexivity. Qed.

Goal 1.23e-2 = 123 / 10000.
Proof. reflexivity. Qed.

(* Les parseurs et printers donnés à Number Notation utilisent alors
   le type Decimal.decimal *)
Print Decimal.decimal.
(* Variant decimal : Set :=
   | Decimal : Decimal.int -> Decimal.uint -> Decimal.decimal
         (* partie entière "." partie fractionnaire *)
   | DecimalExp : Decimal.int -> Decimal.uint -> Decimal.int -> Decimal.decimal
           (* partie entière "." partie fractionnaire "e" exposant *)

  Les entiers peuvent être représentés avec une partie fractionnaire vide :
  Decimal.Nil : Decimal.uint *)

(************************)
(* Nombres hexadécimaux *)
(************************)

(* Depuis Coq 8.12, on dispose également d'une syntaxe hexadécimale *)
Local Open Scope nat_scope.
Check 0x2a.
Check 0X2a.
```

```
(* l'affichage est controlé par le dernier scope ouvert *)
Local Open Scope hex_nat_scope.
Check 42.

(* virgule et exposant (binaire) sont également admis :
   -? ( 0x | 0X ) [0-9a-fA-F]+ ( . [0-9a-fA-F]+ )? ( [pP] [+-]? [0-9]+ )? *)
Local Open Scope Q_scope.
Check 0x0.c.

Goal 0x0.c = 12 / 16.
Proof. reflexivity. Qed.

Check 0x1p8.

Goal 0x1p8 = 256.
Proof. reflexivity. Qed.

(* Les parseurs et printers donnés à Number Notation utilisent alors
   le type HexaDecimal.hexadecimal similaire à Decimal.decimal *)
Print Hexadecimal.hexadecimal.

(* Le choix décimal/hexadécimal est alors disponible
   dans le type somme Number.number *)
Print Number.number.

(*****************)
(* Commentaires *)
(*****************)

(* Des _ peuvent être utilisés comme commentaires
   pour rendre des grandes constantes plus lisibles. *)
Check 1_000_000.

(*******************)
(* Types inductifs *)
(*******************)

(* Pour pouvoir facilement appeler les fonctions de print
   quand un terme est affiché et pouvoir faire des match,
   les types parsés doivent être des inductifs. *)

(* C.f. ci dessus : nat, bool, Q *)

(*****************************)
(* Types inductifs paramétrés *)
(*****************************)
```

```
(* Exemple : listes de booléens *)

Notation blist := (list bool).

Definition uint_of_blist : blist -> Number.uint :=
  fun l =>
    let fix aux l :=
      match l with
      | nil => Decimal.Nil
      | cons true l => Decimal.D1 (aux l)
      | cons false l => Decimal.D0 (aux l)
      end in
    Number.UIntDecimal (aux l).

Definition blist_of_uint : Number.uint -> option blist :=
  fun u =>
    let fix aux l :=
      match l with
      | Decimal.Nil => Some nil
      | Decimal.D1 l =>
          match aux l with None => None | Some l => Some (cons true l) end
      | Decimal.D0 l =>
          match aux l with None => None | Some l => Some (cons false l) end
      | _ => None
      end in
    match u with
    | Number.UIntDecimal l => aux l
    | Number.UIntHexadecimal _ => None
    end.

Number Notation blist blist_of_uint uint_of_blist : bool_scope.

Local Open Scope bool_scope.

Check 1011.
Set Printing All.
Check 1011.
Unset Printing All.
Check cons true (cons false nil).  (* affiché 10 *)
Check cons 1%nat (cons 0%nat nil).  (* pas affiché *)

(* On peut ignorer un paramètre avec _ *)

(* Exemple : affichage de toutes les listes comme leur longueur *)

Definition uint_of_list : list unit -> Number.uint :=
  fun l => Nat.to_num_uint (length l).
```

```
Definition list_of_uint : Number.uint -> list unit :=
  fun u =>
    let fix aux n := match n with O => nil | S n => cons tt (aux n) end in
    aux (Nat.of_num_uint u).

Notation any_list := (list _).
Number Notation any_list list_of_uint uint_of_list : list_scope.

Local Open Scope list_scope.

Check 3.
Set Printing All.
Check 3.
Unset Printing All.
Check cons tt (cons tt nil).  (* affiché 2 *)
Check cons true (cons false nil).  (* aussi affiché 2 *)
Check cons 1%nat (cons 0%nat nil).  (* aussi affiché 2 *)

(************************)
(* Types non inductifs *)
(************************)

(* Certains types numériques ne sont pas inductifs, par exemple les
   réels de la librairie standard. La commande Number Notation,
   permet de parser vers ces types, en utilisant un type inductif
   ad hoc et en établissant une correspondance entre cet inductif
   et des constantes du type concerné. *)

(* Par example pour les réels (c.f., fichier theories/Reals/Rdefinitions.v),
   on déclare le type IR *)
Require Import Reals.
Print IR.
(* Inductive IR : Set :=
   | IRZ : IZ -> IR  (* constantes entières *)
   | IRQ : Q -> IR  (* constantes rationnelles *)
   | IRmult : IR -> IR -> IR  (* multiplication (exposants positifs) *)
   | IRdiv : IR -> IR -> IR. (* division (exposants négatifs) *) *)
Print IZ.
(* Inductive IZ : Set :=
   | IZpow_pos : Z -> positive -> IZ  (* pour encoder les exposants 10^n *)
   | IZ0 : IZ
   | IZpos : positive -> IZ
   | IZneg : positive -> IZ *)

(* la fonction de parse retourne un IR *)
Check of_number.

(* et la fonction de print prend un IR en entrée *)
```

```
Check to_number.

(* Puis la notation est déclarée avec *)
Number Notation R of_number to_number (via IR
  mapping [IZR => IRZ, Q2R => IRQ, Rmult => IRmult, Rdiv => IRdiv,
           Z.pow_pos => IZpow_pos, Z0 => IZ0, Zpos => IZpos, Zneg => IZneg])
  : R_scope.
(* Ici, l'option "via IR" signifie que le type (non inductif) R est
   parsé au travers du type IR, en associant chacun de ses constructeurs
   à une constante de R suivant la liste donnée :

   constructeur | est traduit en la constante
   -------------+------------------------------
   IRZ          | IZR (injection de Z dans R)
   IRQ          | Q2R (injection de Q dans R)
   IRmult       | Rmult (multiplication dans R)
   IRdiv        | Rdiv (division dans R)
   IZpow_pos    | Z.pow_pos (puissance dans Z)
   IZ0          | Z0 (0 dans Z)
   IZpos        | Zpos (positifs dans Z)
   IZneg        | Zneg (négatifs dans Z) *)

Local Open Scope R_scope.

Check 1.2e3.
Set Printing All.
Check 1.2e3.
Unset Printing All.

(************************)
(* Arguments implicites *)
(************************)

(* Certains types inductifs ont des arguments implicites.

   Par exemple, le type Vector.Fin.t encode les entiers naturels plus
   petit qu'une certaine borne n. *)
Require Import Vector.
Print Fin.t.
(* Inductive t : nat -> Set :=
   | F1 : ∀n : nat, Fin.t (S n)
   | FS : ∀n : nat, Fin.t n -> Fin.t (S n)

   Arguments Fin.F1 {n}
   Arguments Fin.FS {n} _

   On note que l'argument n est implicite dans les constructeurs. *)
```

```
(* On utilise alors le type inductif nat (qui n'a pas ce paramètre n)
   come proxy *)

Declare Scope fin_scope.
Delimit Scope fin_scope with fin.
Local Open Scope fin_scope.
Number Notation Fin.t Nat.of_num_uint Nat.to_num_uint (via nat
  mapping [[Fin.F1] => 0, [Fin.FS] => S]) : fin_scope.
(* noter les crochets autour des constructeurs Fin.F1 et Fin.FS :
   leurs arguments implicites seront
   - ignorés en traduisant vers nat avant d'appeler le printer
   - remplacés par des trous _ en traduisant depuis nat après le parse *)

(* Maintenant, 2 est parsé comme Fin.FS (Fin.FS Fin.F1),
   soit @Fin.FS _ (@Fin.FS _ (@Fin.F1 _)) *)
Check 2.

(* qui peut être de type Fin.t 3 par exemple (entiers < 3, soit 0, 1 et 2) *)
Check 2 : Fin.t 3.

(* mais pas de type Fin.t 2 (entiers < 2, soit 0 et 1) *)
Fail Check 2 : Fin.t 2.

(**************)
(* Conclusion *)
(**************)

(* Aujourd'hui dans Coq master, grâce à ces mécanismes, tous les
   anciens parsers/printers numériques sous forme de plugins OCaml ont
   pu être remplacés par des implémentations en Coq.  Cela les rend
   plus facile à écrire, modifier et maintenir mais permet également
   de prouver leur bon comportement, assurant qu'ils n'introduisent
   pas d'incohérence de Pollack. On peut trouver de telles preuves
   dans les fichiers theories/Numbers/Decimal*.v de la librairie
   standard par exemple. *)

(* Plus de détails dans le manuel de Coq :
   https://coq.inria.fr/refman/user-extensions/syntax-extensions.html#numbers-
    and-strings *)
```