École Normale Supérieure de Lyon
Licence d'Informatique Fondamentale, 3e année

Formalisation d'ensembles au plus dénombrables en Coq

Florian Hatat
Stage encadré par Pierre Castéran

Laboratoire Bordelais de Recherche en Informatique — Méthodes formelles

Juin-Juillet 2006

Table des matières

Introduction **2**

1 L'opérateur ϵ d'Hilbert **3**
1.1 Définition de l'opérateur ϵ 3
1.2 ϵ dans le calcul des constructions 3
1.2.1 Cohérence avec le calcul 3
1.2.2 Implémentation choisie en Coq 4
1.2.3 Conséquences sur l'expressivité du calcul 5
1.3 Applications aux développements en Coq 6

2 Formalisation de la dénombrabilité **7**
2.1 Représentation des ensembles dans Coq 7
2.2 Définition de la dénombrabilité en Coq 8
2.3 Validité de la définition 8
2.4 Lemmes de clôture 9
2.5 Exemples et contre-exemples 10

3 Énumération de types de données **11**

Conclusion **13**

Bibliographie **14**

Annexe — Résultats formalisés dans Coq **15**

Introduction

L'assistant de preuve Coq[1], conçu dans le projet Logical, permet d'exprimer des spécifications pour le développement de programmes certifiés, et de manipuler des preuves formelles. Coq implémente un λ-calcul typé et fortement normalisant, le *calcul des constructions inductives*, dans lequel les propositions sont données par le système de types, et où les termes correspondent, d'après l'isomorphisme de Curry-Howard, aux preuves de ces propositions. Ainsi, le cadre naturel du calcul des constructions est la logique intuitionniste ; il est cependant possible d'étendre la théorie en lui ajoutant de nouveaux axiomes, comme l'axiome du tiers-exclu.

Coq travaille dans une logique d'ordre supérieur, qui est indécidable : il ne s'agit donc pas d'un système de preuve automatique, mais d'un assistant qui fournit au développeur un jeu de tactiques pour construire des preuves interactivement. Coq s'assure de leur correction en vérifiant simplement que le terme associé dans le calcul des constructions est bien typé. Il dispose de plus d'un mécanisme d'extraction vers des langages de programmation, ce qui permet de mettre en œuvre un programme en Haskell ou Caml, certifié par Coq.

Cependant, le formalisme du calcul des constructions diffère du formalisme habituel des preuves mathématiques, qui se fondent plutôt sur la théorie des ensembles : cette dernière répond en effet au besoin d'exprimer des spécifications ([5]) compréhensibles et acceptées par tous, tandis que le calcul des constructions permet d'obtenir une théorie constructive, et vérifiable automatiquement. Il est alors nécessaire de porter les concepts de la théorie des ensembles dans le calcul des constructions.

J'ai ainsi implémenté au cours de ce stage la notion d'*ensembles dénombrables* dans le calcul des constructions : il s'agissait de choisir un modèle pour exprimer la dénombrabilité dans Coq, et de prouver les principaux résultats obtenus dans la théorie des ensembles. Ce travail m'a de plus conduit à travailler avec l'opérateur ϵ d'Hilbert, qui permet notamment d'approcher en Coq un mode de développement similaire aux raisonnements mathématiques usuels.

[1] L'ensemble des sources est disponible sur : `http://coq.inria.fr/`

Chapitre 1

L'opérateur ϵ d'Hilbert

1.1 Définition de l'opérateur ϵ

L'opérateur ϵ [1] est à la base d'une théorie logique développée par Hilbert afin de formaliser les raisonnements mathématiques. Hilbert introduit l'expression :

$$\epsilon x.M$$

où M est une proposition et x une variable, pour noter *un certain* x qui satisfait M. Par exemple le terme $\epsilon x.(x = a \vee x = b)$ représente un élément de $\{a, b\}$, sans qu'il ne soit cependant possible de déterminer s'il s'agit de a ou de b.

L'introduction des ϵ-termes est accompagnée de l'ajout de leur règle d'élimination : $M \to M[x := \epsilon x.M]$.

Ces termes sont à la base du formalisme développé par Hilbert : celui-ci ne contient pas de quantificateurs, mais il est possible de les définir comme des ϵ-termes :

$$\begin{aligned} \exists x, P(x) &\equiv P(\epsilon x.P(x)) \\ \forall x, P(x) &\equiv P(\epsilon x.\neg P(x)) \end{aligned}$$

En se plaçant dans la logique classique, on peut vérifier la cohérence de cette définition avec Coq, qui contient déjà des quantificateurs définis sans ϵ, en montrant que ces propositions sont équivalentes.

1.2 ϵ dans le calcul des constructions

1.2.1 Cohérence avec le calcul

L'introduction de l'opérateur ϵ dans le calcul des constructions n'est possible qu'avec une légère adaptation. En effet, si l'on copiait sa définition d'origine, l'opérateur ϵ deviendrait un terme dont le type serait :

$$\forall A, (A \to \mathit{Prop}) \to A$$

Cependant en Coq il existe des types non habités : on peut par exemple définir le type inductif *empty* qui ne possède pas de constructeur :

Inductive *empty* : *Set* := .

Il est impossible de construire un terme possédant le type *empty*, or d'après le type donné pour ϵ, le terme :

$$epsilon\ (\lambda(x : empty).x = x)$$

posséderait justement le type *empty*. Ceci permettrait de dériver ensuite *False*, et donc de montrer que la théorie devient incohérente.

C'est pourquoi l'implémentation d'ϵ en Coq[1] demande en plus un témoin indiquant que le type A est habité :

$$epsilon : \forall A, A \to (A \to Prop) \to A$$

Avec ce type, l'équivalent en Coq de l'ϵ-terme qui désigne un élément de l'ensemble $\{a, b\}$ peut alors s'écrire :

$$epsilon\ t\ (\lambda x.x = a \vee x = b)$$

Le témoin t est un élément quelconque du même type que a et b.

1.2.2 Implémentation choisie en Coq

L'opérateur ϵ est défini dans le module `ClassicalEpsilon`, écrit par Hugo Herbelin, introduit dans la bibliothèque standard de Coq à partir de la version 8.1. Ce module utilise une forme de *l'axiome du choix* pour construire ϵ :

Axiom *constructive_indefinite_description* :
∀ (*A* : *Type*) (*P* : *A*→*Prop*), (*ex P*) → { *x* : *A* | *P x* }.

Cet axiome ajoute au calcul des constructions une fonction qui permet, à partir d'un prédicat P et d'une preuve d'existence d'un élément satisfaisant P, de *construire* un objet x du calcul qui satisfait effectivement P.

Le calcul des constructions distingue en effet deux quantificateurs existentiels :

- le prédicat *ex*, de type $\forall A : Type, (A \to Prop) \to Prop$, permet d'exprimer l'existence d'un élément satisfaisant un prédicat, mais ne donne aucune possibilité de construire cet élément. Ce quantificateur est utilisé dans les propositions de la forme : $\exists x : T, P\,x$;
- le type *sigT*, de type $\forall A : Type, (A \to Type) \to Type$, fournit en plus un moyen de construire effectivement l'élément. On note alors pour ce quantificateur : $\{x : T \mid P\,x\}$.

L'axiome *constructive_indefinite_description* identifie ces deux notions, en permettant de construire un objet uniquement à partir d'une preuve de son existence. Cette construction n'est cependant pas effective, car les termes créés contiennent un appel à l'axiome, qui n'est pas extractible dans un langage de programmation.

Cet axiome permet ensuite de dériver le théorème :

Theorem *classical_indefinite_description* :

[1] L'écriture des ϵ-termes diffère entre le formalisme d'Hilbert et la syntaxe de Coq : pour distinguer ces deux contextes, nous utiliserons dans la suite la notation $\epsilon x.M$ pour les termes de l'ϵ-calcul, et la notation *epsilon t P* pour les termes de Coq.

pour les ϵ-termes :

$$(\forall x, P(x) \leftrightarrow Q(x)) \rightarrow \epsilon x.P = \epsilon x.Q$$

qui indique que l'opérateur retourne toujours le même élément pour deux propositions équivalentes.

1.3 Applications aux développements en Coq

Le calcul des constructions est fortement normalisant, ce qui impose deux restrictions principales sur les fonctions que l'on peut définir en Coq : tous les calculs doivent se terminer, et toute fonction est nécessairement totale. Il est cependant possible d'utiliser l'opérateur ϵ pour contourner cette deuxième restriction, comme dans l'exemple suivant, où l'on définit la division par deux, uniquement sur les entiers pairs :

$$\mathsf{Definition}\ \mathit{div2e}\ (n : \mathit{nat}) := \mathit{epsilon}\ 0\ (\lambda m.m + m = n).$$

La notion de fonction partielle est obtenue grâce à la spécification d'*epsilon* : en effet l'utilisation de la règle d'élimination générera un but demandant une preuve de l'existence de l'entier m. Ce but est simplement improuvable dans les cas où la fonction n'est pas définie. Le fichier `http://www.labri.fr/perso/casteran/Half.v` utilise *epsilon* sur ce modèle pour formaliser en arithmétique la division par 2, et le logarithme exact en base 2.

Le développement sur les ensembles dénombrables utilise des relations entre les ensembles pour comparer leurs cardinaux, là où la théorie des ensembles utilise plutôt des fonctions partielles. L'équivalence entre ces deux approches peut s'établir avec ϵ, qui permet de transformer une relation R, de type $A \rightarrow B \rightarrow \mathit{Prop}$, en une fonction partielle f, de type $A \rightarrow B$:

$$f(a) := \mathit{epsilon}\ b_0\ (\lambda b.a\ R\ b)$$

Cet exemple montre que l'utilisation d'*epsilon* permet de retrouver l'axiome du choix fonctionnel, qui peut s'énoncer ainsi :

$$(\forall x, \exists y, x\ R\ y) \rightarrow \exists f, \forall x, x\ R\ f(x)$$

On pourrait même interpréter le premier argument d'*epsilon*, le témoin b_0, comme un élément qui indexerait le choix effectué par *epsilon*. Cet aspect se retrouve dans le fait que deux termes *epsilon* b_0 P et *epsilon* b_1 P, utilisant deux témoins différents b_0 et b_1, sont totalement indépendants : il est impossible par exemple de prouver qu'ils sont égaux, ou même différents, sauf si l'on peut prouver une propriété d'unicité sur le prédicat P.

Le module `Epsilon` contient plusieurs définitions implémentant l'axiome du choix à l'aide de l'opérateur ϵ, ainsi que des tactiques permettant de manipuler les fonctions partielles ainsi définies dans une preuve.

Chapitre 2

Formalisation de la dénombrabilité

2.1 Représentation des ensembles dans Coq

Pour définir la notion d'ensembles en Coq, il serait possible d'utiliser l'approche axiomatique décrite dans [6], en ajoutant dans le calcul des constructions les axiomes de Zermelo-Fraenkel. Cependant cette méthode peut s'avérer dangereuse, car il faut alors vérifier la cohérence des nouveaux axiomes avec le calcul : la FAQ [7] de Coq mentionne quelques axiomes qui peuvent être ajoutés sans créer d'incohérence — comme l'axiome du tiers-exclu —, mais également des axiomes qui rendent la théorie incohérente.

On préfère donc définir les ensembles à partir de constructions déjà existantes. Ainsi la bibliothèque *Ensembles* de Coq définit des ensembles d'objets, pris sur un type univers U, en identifiant un ensemble A à son prédicat caractéristique :

$$\mathit{Ensemble}\ A \equiv U \to \mathit{Prop}$$

La plupart des axiomes de la théorie des ensembles sont alors vérifiés, à l'exception de l'axiome d'extensionnalité, qui est ajouté au calcul par la bibliothèque *Ensembles* :

$$\forall A, B, (A \subset B \wedge B \subset A) \to A = B$$

Les ensembles que l'on obtient diffèrent légèrement de la notion habituelle, à cause des contraintes de type qui impliquent par exemple l'existence de plusieurs ensembles vides, car ces ensembles sont typés. Cependant ces contraintes permettent de s'affranchir des paradoxes de la théorie des ensembles, tels que le paradoxe de Russel. En effet dans le cas de ce dernier, la relation d'appartenance est définie comme un prédicat du type :

$$\forall U : \mathit{Type}, U \to \mathit{Ensemble}\ U \to \mathit{Prop}$$

Les termes de la forme $e \in e$ ne sont donc pas des termes bien typés car e devrait être de type U et $U \to \mathit{Prop}$.

2.2 Définition de la dénombrabilité en Coq

Plusieurs propositions équivalentes peuvent définir la dénombrabilité dans la théorie des ensembles :

$$\begin{aligned} & A \text{ dénombrable} \\ \iff \quad & \exists f : A \to \mathbb{N}, f \text{ injective} \\ \iff \quad & \exists g : \mathbb{N} \to A, g \text{ surjective} \\ \iff \quad & \exists P \subseteq \mathbb{N}, \exists f : A \to P, f \text{ bijective} \\ \iff \quad & \exists k \in \mathbb{N} \cup \{\infty\}, \exists f : A \to [\![0, k[\![, f \text{ bijective} \end{aligned}$$

La définition que l'on choisit en Coq détermine directement la difficulté des preuves qu'il faudra écrire pour obtenir certains résultats. Ainsi la dernière caractérisation, qui utilise une bijection vers un segment initial de $\mathbb{N}$, impose en général de distinguer dans chaque preuve les cas où P est fini ou infini, ce qui complique rapidement les preuves en créant de nombreux cas à traiter.

Nous avons donc choisi une définition proche de la caractérisation par des applications injectives. Si A est un ensemble d'objets de type U, on définit :

$$A \text{ dénombrable} \iff \exists R : U \to \mathbb{N} \to \mathit{Prop}, R \text{ injective}$$

L'utilisation d'une relation R à la place d'une fonction est due au fait que toute fonction dans le calcul des constructions est a priori totale, donc pour définir une application injective sur A, il faudrait définir une fonction $f : U \to \mathbb{N}$ sur le type U en entier. Cette restriction est trop contraignante, car si l'on voulait prouver par exemple qu'une partie de $\mathbb{R}$ est dénombrable, il serait nécessaire de définir f sur tout $\mathbb{R}$. Une relation permet au contraire d'obtenir une notion de fonction partielle.

Plus formellement, nous avons défini un prédicat *denumerable* en Coq, qui est satisfait par un ensemble A dès qu'il existe une relation R telle que :

(i) $\quad \forall x : U,\ x \in A \to \exists n : \mathbb{N},\ x\, R\, n$

(ii) $\quad \forall n : \mathbb{N},\ \forall x\, x' : U,\ x \in A \to x' \in A \to x\, R\, n \to x'\, R\, n \to x = x'$

Nous imposons ainsi que R soit définie et injective sur A, mais R n'est pas nécessairement une relation fonctionnelle : un élément x, de type U, peut avoir plusieurs images. Nous avons vu cependant en 1.3 que l'utilisation de l'opérateur ϵ permet de passer d'une relation à une fonction partielle, ce qui garantit que cette définition est cohérente avec la définition usuelle de la dénombrabilité.

2.3 Validité de la définition

L'opérateur ϵ permet de faire le lien entre la définition que nous avons donnée en Coq d'un ensemble dénombrable, et la définition usuelle de la théorie des ensembles. De plus, le théorème *denumerable_surj* montre l'équivalence avec la caractérisation par des relations fonctionnelles et surjectives, c'est-à-dire vérifiant :

(i) $\forall x : U,\ x \in A \to \exists n : \mathbb{N},\ n\,R\,x$

(ii) $\forall n : \mathbb{N},\ \forall x\,x' : U,\ x \in A \to x' \in A \to n\,R\,x \to n\,R\,x' \to x = x'$

J'ai de plus relié la dénombrabilité à la notions d'ensembles finis, définie dans la bibliothèque *Finite_sets*, ainsi qu'à la notion de cardinalité inférieure, formalisée en Coq par Hugo Herbelin pour prouver le théorème de Schröder-Bernstein.

2.4 Lemmes de clôture

La plupart des résultats habituels de stabilité de la classe des ensembles dénombrables se prouvent assez facilement en Coq :

- l'union de deux ensembles dénombrables est dénombrable, ce résultat est prouvé en Coq par le théorème *denumerable_union* ;
- l'intersection de deux ensembles dénombrables est dénombrable (théorème *denumerable_intersection*) ;
- une partie d'un ensemble dénombrable est dénombrable (théorème *denumerable_inclusion*).

Une preuve, plus technique, montre que l'union dénombrable d'ensembles dénombrables est dénombrable (théorème *denumerable_union_qcq*). Cette preuve est similaire à la preuve que le produit $A \times B$ est dénombrable dès que A et B le sont, et utilise la bijection :

$$\begin{array}{rcl} \mathbb{N} \times \mathbb{N} & \to & \mathbb{N} \\ (p, q) & \mapsto & \dfrac{(p+q)(p+q+1)}{2} + q \end{array}$$

Cette fonction numérote les couples (p, q), en partant du couple $(0, 0)$ et en suivant les diagonales, identifiées par $p + q$ constant. Sa réciproque K peut se définir par récurrence :

```
Fixpoint K (n : nat) : nat * nat :=
  match n with
    0 ⇒ (0, 0)
    | S n ⇒ match K n with
      (0, m) ⇒ (S m, 0)
      | (S n, m) ⇒ (n, S m)
    end
  end.
```

K induit un ordre $<_K$ sur $\mathbb{N} \times \mathbb{N}$, que l'on peut exprimer par :

$$(p, q) <_K (p', q') \iff (p + q) < (p' + q') \vee (p + q = p' + q' \wedge q < q')$$

Cet ordre est bien fondé, car il s'agit de l'image inverse de l'ordre lexicographique par la fonction $(p, q) \mapsto (p + q, q)$:

$$(p, q) <_K (p', q') \iff (p + q, q) <_{lex} (p' + q', q')$$

La bonne fondation de $<_K$ permet alors de prouver facilement par induction tous les lemmes dont dépend la preuve de *denumerable_union_qcq*, notamment le fait que K est injective, dont découle directement l'injectivité de la relation énumérant l'union dénombrable.

2.5 Exemples et contre-exemples

L'application immédiate du prédicat *denumerable* permet de prouver trois premiers exemples :

- l'ensemble des entiers naturels est dénombrable ;
- toute partie de $\mathbb{N}$ est dénombrable ;
- $\mathbb{Z}$ est dénombrable.

Il était également intéressant de donner une preuve de non-dénombrabilité, en formalisant l'argument diagonal de Cantor, qui permet de prouver que l'ensemble des suites sur $\mathbb{N}$ n'est pas dénombrable. Ce dernier est pris comme l'ensemble de tous les habitants du type $\mathbb{N} \to \mathbb{N}$:

```
Definition Suites := Full_set (nat → nat).
```

L'argument mathématique habituel consiste à supposer qu'il existe une relation R qui énumère l'ensemble de ces suites, puis à construire une nouvelle suite qui diffère de chaque suite énumérée par R. Cette construction devient immédiate en Coq en utilisant l'opérateur ι :

```
Definition diagonal_seq (n : nat) :=
  S (iota (fun _ ⇒ 0) (fun u ⇒ R n u) n).
```

On dérive ensuite la contradiction voulue, à l'aide de l'élimination d'ϵ.

Cet exemple montre ainsi que l'utilisation de l'opérateur ϵ permet d'obtenir un développement en Coq qui n'est presque qu'une transcription de l'argument diagonal formulé dans la théorie des ensembles.

Chapitre 3

Énumération de types de données

Les types de données inductifs sont dénombrables, car leurs habitants sont construits par un nombre fini d'applications des constructeurs : cette constatation m'a conduit à tenter de construire des énumérations certifiées, qui pourront faire l'objet d'une extraction vers un langage de programmation comme Caml. Ces énumérations sont des fonctions partielles de types $\mathbb{N} \to \mathit{option}\, D$ et $D \to \mathit{option}\, \mathbb{N}$, bijectives et réciproques l'une de l'autre.

La technique utilisée est inspirée du formalisme mis en place dans [4], qui décrit les définitions des types inductifs en termes d'unions disjointes, notées $+$, et de produits cartésiens, notés $\times$, sur des ensembles. Le type `bintree` des arbres binaires est par exemple défini comme le plus petit type vérifiant l'expression :

$$\texttt{bintree} := \texttt{leaf} + \texttt{bintree} \times \texttt{bintree}$$

Le constructeur `leaf` s'assimile simplement à l'ensemble $\{\emptyset\}$.

Il est alors naturel d'écrire des fonctionnelles qui composent des fonctions d'énumération pour créer des énumérations pour le produit cartésien ou l'union disjointe. J'ai par exemple créé une fonctionnelle $\mathcal{F}_{du}$ qui prend en argument deux fonctions d'énumération f et g, associées respectivement à deux ensembles A et B, et qui renvoie une fonction d'énumération h pour l'union disjointe $A \cup B$. $\mathcal{F}_{du}$ a pour type[1], dans un contexte où A et B sont deux ensembles avec l'hypothèse qu'ils sont disjoints :

```
{f | fun_enumerates f A} → {g | fun_enumerates g B} →
  {h | fun_enumerates h (Union _ A B)}
```

On définit de même des fonctionnelles $\mathcal{F}_{cp}$ et $\mathcal{F}_{iu}$, respectivement pour le produit cartésien et l'union infinie dénombrable. Dès lors, l'énumération d'un type inductif comme `bintree` se définit comme le point fixe :

```
bintree_enum =
  F_du (fun _ ⇒ leaf)
    (F_cp bintree_enum bintree_enum)
```

[1] Le prédicat *fun_enumerates* est défini en annexe, p.17.

Ce procédé a l'avantage de donner une méthode générale pour énumérer n'importe quel type inductif. Cependant pour obtenir une fonction d'énumération satisfaisant une équation de point fixe de ce type, il est nécessaire d'utiliser un argument de bonne fondation afin de garantir la terminaison du calcul, ce que je n'ai eu le temps de faire pendant ce stage que pour le seul exemple des arbres binaires :

```
Inductive bintree : Set :=
  leaf : bintree
  | node : bintree → bintree → bintree.

Definition bintree_enum_cert :
  {bintree_enum : nat → option bintree | fun_enumerates
     bintree_enum (Full_set bintree)}.
```

Si l'on entrevoit donc une méthode d'énumération de tout un type, il devient légitime de s'interroger sur l'énumération d'une partie seulement de ce type — comme les arbres binaires de recherche par exemple, dans le type `bintree` —. Je n'ai pas eu le temps d'effectuer de formalisation en Coq de cette idée, cependant il existe plusieurs pistes pour arriver à ce résultat : on peut notamment se ramener à l'énumération d'un type inductif en entier si l'on arrive à exprimer par induction le prédicat caractéristique des objets que l'on veut énumérer. Le point fixe n'est alors plus créé à partir de la structure inductive du type, mais d'après la structure du prédicat caractéristique.

Conclusion

Ce stage m'a permis de me familiariser avec le fonctionnement de l'assistant de preuve Coq, mais également de découvrir certains concepts fondant d'autres assistants tels que Isabelle/HOL ou le système B.

Le développement sur les ensembles dénombrables que j'ai écrit durant ce stage a de plus permis de résoudre quelques résultats qui étaient demeurés admis dans le développement sur les ordinaux de Pierre Castéran. Mon travail a pris place dans le projet ANR A3PAT[2], et a été inclus dans le répertoire `denumerable/` du développement d'Evelyne Contéjean et de Pierre Castéran sur RPO et les ordinaux. Les sources complètes de cette contribution à Coq 8.1 sont disponibles à l'adresse : `http://www.labri.fr/~casteran/Cantor/Kantor.tar.gz`

De plus, l'utilisation de l'opérateur epsilon ouvre une voie intéressante pour les développements en Coq : l'axiomatisation d'epsilon en Coq est récente, et mérite d'être explorée car elle ne présente aucune contradiction avec le noyau intuitionniste du calcul des constructions, et elle permet d'utiliser un langage mathématique usuel dans les développements. L'écriture de modules utilisant epsilon permet ainsi d'une part d'exprimer des spécifications dans ce langage usuel, et de réaliser d'autre part des développements de programmes effectifs à l'intérieur du même outil.

[2]Assister Automatiquement les Assistants de Preuve Avec des Traces, `http://www3.iie.cnam.fr/~urbain/a3pat/`

Bibliographie

[1] Jeremy Avigad et Richard Zach. The epsilon calculus. Dans Edward N. Zalta, rédacteur, *The Stanford Encyclopedia of Philosophy*. Summer 2002. URL `http://plato.stanford.edu/archives/sum2002/entries/epsilon-calculus/`

[2] John L. Bell. Hilbert's epsilon-operator in intuitionistic type theories. *Math. Log. Q.*, 39 :323–337, 1993.

[3] Yves Bertot et Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art : The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
URL `http://www.labri.fr/publications/l3a/2004/BC04`

[4] William H. Burge. *Recursive Programming Techniques*. The Systems Programming Series. Addison-Wesley, 1975.

[5] Gilles Dowek. Théorie des types.
URL `http://www.lix.polytechnique.fr/~dowek/Cours/theories_des_types.ps.gz`

[6] Paul R. Halmos. *Naive Set Theory*. Undergraduate Texts in Mathematics. Springer Verlag, 1974.

[7] Hugo Herbelin, Florent Kirchner, Benjamin Monate, et Julien Narboux. Faq about coq.
URL `http://coq.inria.fr/doc/faq.html`

Annexe — Résultats formalisés dans Coq

Définitions importées de `PartialFun.v`

```
Definition rel_domain :=
  ∀ a, In DA a → ∃ b, Rf a b.
Definition rel_codomain :=
  ∀ a b, In DA a → Rf a b → In DB b.
Definition rel_functional :=
  ∀ a b b', In DA a → Rf a b → Rf a b' → b = b'.
Definition rel_onto :=
  ∀ b, In DB b → ∃ a, In DA a ∧ Rf a b.
Definition rel_inj :=
  ∀ a a' b, In DA a → In DA a' →
    Rf a b → Rf a' b → a = a'.

Inductive rel_injection : Prop :=
  rel_inj_i : rel_domain DA R →
    rel_codomain DA DB R →
    rel_inj DA R →
    rel_injection.

Inductive rel_surjection : Prop :=
  rel_surj_i : rel_codomain DA DB R →
    rel_functional DA R →
    rel_onto DA DB R →
    rel_surjection.
```

Résultats sur la dénombrabilité

```
(** A formalization of denumerable sets. *)

Section Denumerable.

Section Definitions.
  Variable U : Type.
  Variable A : Ensemble U.
```

```
  Let Dnat : Ensemble nat := Full_set nat.

  Definition rel_numbers (R: GRelation U nat) :=
    rel_injection A Dnat R.

  (** Predicate for relations which enumerate A. *)
  Definition rel_enumerates (R : GRelation nat U) :=
    rel_surjection Dnat A R.

  Definition denumerable : Prop :=
    ∃ R, rel_numbers R.

  Theorem denumerable_surj :
    denumerable ↔ ∃ R, rel_enumerates R.
End Definitions.

Variable U : Type.

Theorem denumerable_union (E : Ensemble U) (F : Ensemble
  U) :
  denumerable E → denumerable F → denumerable (Union U E
    F).

Theorem denumerable_inclusion (E : Ensemble U) (F :
  Ensemble U) :
  denumerable E → Included _ F E → denumerable F.

Section Infinite_union.
  Variable A : Ensemble (Ensemble U).
  Definition Infinite_union (x : U) : Prop :=
    ∃ b, In A b ∧ In b x.

  Theorem denumerable_union_qcq :
    denumerable A →
      (∀ b : Ensemble U, In A b → denumerable b) →
        denumerable Infinite_union.
End Infinite_union.

Lemma denumerable_singleton (x : U) :
  denumerable (Singleton _ x).

Section Denumerable_seq_range.
  Definition seq_range (f : nat → U) :=
    fun x ⇒ ∃ n : nat, f n = x.

  Lemma seq_range_denumerable :
    ∀ f, denumerable (seq_range f).
End Denumerable_seq_range.

Section Denumerable_bijection.
```

```
  Variable V : Type.

  Variable A : Ensemble U.
  Variable B : Ensemble V.
  Variable g : U → V.

  Hypothesis g_bij : fun_bijection A B g.

  Lemma denumerable_bij_fun :
    denumerable A → denumerable B.

  Lemma denumerable_bij_funR :
    denumerable B → denumerable A.
End Denumerable_bijection.

Theorem denumerable_finite :
  ∀ A : Ensemble U, Finite _ A → denumerable A.
End Denumerable.
```

Énumérations effectives

```
Inductive segment_bound : Set :=
  all_nats : segment_bound
  | less_than : nat → segment_bound.

Definition nat_segment (b : segment_bound) : Ensemble nat
    := match b with
    all_nats ⇒ (fun _ ⇒ True)
    | less_than p ⇒ (fun i ⇒ i < p)
  end.

Definition fun_domain (b : segment_bound)
    (X : Ensemble D) (f : nat → option D) :=
  ∀ n, In (nat_segment b) n → ∃ x, In X x ∧ f n = Some x.

Definition fun_injection (b : segment_bound) (X :
    Ensemble D)
  (f : nat → option D) :=
  ∀ n n', In (nat_segment b) n → In (nat_segment b) n' →
    f n = f n' → n = n'.

Definition fun_surjection (b : segment_bound) (X :
    Ensemble D)
  (f : nat → option D) :=
  ∀ x, In X x → ∃ n, In (nat_segment b) n ∧ f n = Some x.

Inductive fun_enumerates (b: segment_bound)
    (X : Ensemble D) (f : nat → option D) : Prop :=
  is_bijection : fun_domain b X f → fun_injection b X f →
    fun_surjection b X f → fun_enumerates b X f.
```