Preuve de programmes d'énumération avec Why3

Alain Giorgetti et Rémi Lazarini
FEMTO-ST Institute, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France

Résumé

L'énumération est une technique élémentaire de génération automatique de données pour le test du logiciel. Cet article présente une spécification formelle de programmes d'énumération de données stockées dans des tableaux d'entiers, pour leur vérification déductive avec la plateforme Why3. L'approche est illustrée par l'exemple de l'énumération de toutes les permutations d'une taille donnée, dans l'ordre lexicographique.

1 Introduction

Les outils de vérification formelle de programmes ont désormais atteint une maturité qui permet de les utiliser dans l'industrie du logiciel critique. Ces implémentations complexes laissent cependant planer des doutes sur leur propre correction. Plusieurs travaux contribuent à rétablir la confiance en ces outils, en certifiant certains de leurs composants. Par exemple, en tant que composant d'un environnement de test, un solveur de contraintes a été certifié avec l'assistant de preuve Coq [CDG12]. Le test automatique fondé sur les propriétés a été formalisé en Coq [PHD+15] pour certifier des générateurs aléatoires de données de test.

Dans cette lignée, ce travail est une contribution à la certification d'outils de test exhaustif borné (BET, pour *Bounded-Exhaustive Testing*). Le BET [SYC+04] automatise le test unitaire en générant toutes les données possibles d'entrée de la fonction sous test, jusqu'à une taille donnée. Quoique cette méthode soit limitée aux données de petite taille, sa pertinence est reconnue : "*a large portion of faults is likely to be revealed already by testing all inputs up to a small scope*" [JD96]. Fournissant toujours des contre-exemples de taille minimale, le BET facilite le déboguage. Il est complémentaire de méthodes adaptées aux données de plus grande taille, comme la génération aléatoire.

Le BET s'applique aux données munies d'une taille telle qu'il existe un nombre fini de données différentes de chaque taille. Ces données sont appelées *structures combinatoires*. Nous souhaitons spécifier et vérifier formellement diverses propriétés des programmes qui implémentent des algorithmes d'énumération de ces structures combinatoires, lorsqu'elles sont stockées dans un tableau d'entiers bornés soumis à des contraintes structurelles, telles que l'unicité des valeurs. Nous traitons ici l'exemple significatif d'un tableau de longueur n qui stocke les valeurs d'une permutation des n premiers entiers naturels.

La combinatoire énumérative propose de nombreux algorithmes de génération exhaustive bornée, plus ou moins difficiles à spécifier et vérifier formellement. Nous étudions ici les algorithmes dits de *génération lexicographique*, qui énumèrent tous les tableaux de même taille selon l'ordre lexicographique, défini comme suit :

Définition 1 (Ordre lexicographique) *Soit A un ensemble muni d'un ordre strict $<$ (relation binaire irréflexive et transitive). On appelle* ordre lexicographique (strict, induit par $<$) *la relation binaire sur*

les tableaux à valeurs dans A, notée $\prec$, telle que, pour tout entier $n \geq 0$ et pour tous les tableaux a et b de longueur n dont les éléments sont dans A, on a $a \prec b$ si et seulement s'il existe un indice i ($0 \leq i \leq n-1$) tel que $a[j] = b[j]$ pour tout j entre 0 et $i-1$ inclus et $a[i] < b[i]$.

La relation binaire $\prec$ ainsi définie est un ordre strict, qui est total si l'ordre sur A est total.

L'énumération est une forme particulière du concept général d'itération en programmation. Filliâtre et Pereira [FP16a, FP16b] ont spéficié formellement le processus d'itération lorsqu'il parcourt des données stockées en mémoire. Nous complétons ce travail en spécifiant une autre forme d'itération, appelée *énumération* ou *génération exhaustive*, qui consiste à produire chaque donnée au fur et à mesure, à la volée, avec pas ou peu de stockage en mémoire des précédentes données produites.

Par analogie avec la classification des itérateurs à grands pas et à petits pas [FP16a], nous considérons deux classes de générateurs exhaustifs. Un *générateur à grands pas* garde le contrôle de l'itération, en produisant lui-même toutes les structures combinatoires d'une même famille, tout en appliquant le même traitement à chacune d'elles. Il est souvent implémenté à partir d'un *générateur à petits pas* dont chaque appel construit une nouvelle structure de la même famille, à partir d'une quantité d'information limitée, typiquement la précédente structure construite par ce générateur. Un intérêt d'un générateur à petits pas est qu'il laisse le contrôle au code client, lui permettant par exemple de paralléliser l'énumération.

Nous souhaitons prouver formellement quatre propriétés de ces générateurs. La *correction* est la propriété que chaque structure générée satisfait la contrainte structurelle attendue (tableau borné, permutation, ...). La *complétude* exige que le générateur produise toutes les structures d'une même taille donnée. La *terminaison* exige l'arrêt de l'énumération. L'*absence de doublons* exige que chaque donnée ne soit générée qu'une seule fois. Nous considérons ici que ces deux dernières propriétés sont des conséquences de la propriété de *progression*, selon laquelle la donnée produite par le générateur à petits pas est strictement supérieure à sa donnée d'entrée, pour l'ordre lexicographique de la définition 1.

Plusieurs générateurs séquentiels de tableaux structurés ont été implémentés en C et spécifiés en ACSL [GGP15]. Leur correction et leur progression ont été prouvées automatiquement avec le greffon WP de la plateforme Frama-C, mais pas leur complétude, dont les conditions de vérification sont plus complexes. Pour les simplifier, nous adaptons ces générateurs au langage WhyML de la plateforme Why3 [BFM+18] et à sa structure de tableaux mutables. Why3 est une plateforme de vérification déductive qui permet d'utiliser des prouveurs automatiques (comme Alt-Ergo [Alt18]), mais aussi des assistants de preuve (comme Coq). Son mécanisme d'extraction de générateurs OCaml corrects par construction permet par exemple d'utiliser ces générateurs dans un outil de test exhaustif borné de conjectures Coq [DG18].

Les contributions de cet article sont une spécification formelle de la notion de générateur lexicographique en WhyML (partie 2) et une discussion sur la vérification déductive de leurs propriétés (partie 3). La partie 4 conclut cette étude. Une version étendue de cet article et le code présenté sont téléchargeables ici : `http://members.femto-st.fr/alain-giorgetti/fr/ressources`.

2 Spécification des générateurs lexicographiques

Un *générateur lexicographique* à grands pas énumère toutes les structures combinatoires d'une même famille (comme les permutations) dans l'ordre lexicographique $\prec$ (définition 1). Nous étudions le cas où il procède par itération d'un générateur lexicographique à petits pas qui construit la structure b à partir de la structure a qui la précède immédiatement selon l'ordre $\prec$. Nous spécifions successivement en WhyML le générateur à grands pas, la propriété de correction et les fonctions d'initialisation

et de passage au suivant du générateur à petits pas. Nous illustrons ces spécifications avec l'exemple fil-rouge d'un générateur lexicographique de permutations. Pour tout entier naturel n, la permutation p sur $[0..n-1]$ est représentée en WhyML par un tableau d'entiers mutable `a`, de type `(array int)` et de longueur `a.length` $= n$, tel que `a[`i`]` $= p(i)$ pour $0 \leq i < n$.

Génération de toutes les structures. Les générateurs sont spécifiés dans un style impératif, avec un état modifié et des boucles pour itérer. L'état des générateurs est stocké dans une structure de données appelée *curseur*, définie en WhyML par le type suivant :

```
type cursor = { current: array int; mutable new: bool; }
```

Cette structure est une variante de la structure de curseur proposée par Filliâtre et Pereira pour l'itération [FP16b]. Le curseur contient un champ `current` qui stocke la dernière structure générée et un champ `new` qui prend la valeur `true` si la structure stockée dans le champ `current` est nouvelle, c'est-à-dire qu'elle n'a pas encore été traitée. Le type de la structure `current` est ici un tableau d'entiers mais la spécification peut être adaptée à d'autres types.

Le générateur à grands pas

```
let gen (c: cursor) = let f = ... in while c.new do f c.current; next c done
```

doit être appelé avec un curseur `c` dont le champ `current` stocke une première structure. Le corps de la boucle applique un traitement (une fonction `f`) à chaque nouvelle structure, puis génère la structure suivante en appelant le générateur à petits pas `next` (présentée plus loin) qui modifie le curseur par effets de bord. La boucle se répète tant qu'une structure différente est générée par la fonction `next`, ce qui est caractérisé par l'égalité `c.new = true`.

Propriété de correction. La correction est spécifiée par le prédicat

```
predicate sound (c: cursor) = is_X c.current
```

à partir d'un prédicat `is_X` caractéristique de la famille `X` de structures générée. Par exemple, la famille `permut` des permutations est définie comme une endofonction injective, par le prédicat

```
predicate is_permut (a:array int) = (range a) ∧ (injective a)
```

où `(range a)` spécifie que le tableau `a` est à valeurs dans $[0..$`a.length` $- 1]$ et `(injective a)` spécifie l'injectivité de la fonction représentée par le tableau `a`. Nous avons plus généralement défini l'injectivité pour le type polymorphe `(array 'a)`, en spécialisant un prédicat analogue défini dans la librairie standard de Why3 pour le type polymorphe `(map 'a 'b)` des tableaux associatifs.

Création et initialisation du curseur. Les générateurs `gen` et `next` prennent en paramètre d'entrée un curseur `c` qui doit être construit et initialisé par une fonction

```
val create_cursor (n: int) : cursor
  requires { n ≥ 0 }
  ensures  { result.new → sound result }
```

qui prend en paramètre la taille n des structures à générer. Sa précondition impose une taille positive ou nulle. Sa postcondition exige que le champ `current` du curseur construit contienne une structure de la famille `X`, si son champ `new` contient la valeur `true` (ce champ doit valoir `false` s'il n'existe aucune structure de taille n).

Par exemple, la fonction `create_cursor` du listing 1 construit un curseur contenant la permutation identité ($p(i) \leftarrow i$), à l'aide d'une boucle `for` spécifiée par un invariant `(is_id` p i`)` qui exige que la partie $p[0..i-1]$ du tableau p représente l'identité. Ceci permet de démontrer la postcondition de correction de ce curseur. La seconde postcondition indique que cette permutation identité existe et qu'elle est nouvelle pour toute taille $n \geq 0$. Ces postconditions sont plus précises que dans le cas général car il existe une permutation pour toute taille $n \geq 0$.

```
predicate sound (c: cursor) = is_permut c.current
predicate is_id (a:array int) (n:int) =
  ∀ i:int. 0 ≤ i < n → a[i] = i

let create_cursor (n: int) : cursor
  requires { n ≥ 0 }
  ensures  { sound result }
  ensures  { result.new }
= let p = make n 0 in
  for i = 0 to p.length - 1 do
    invariant { 0 ≤ i ≤ p.length }
    invariant { is_id p i }
    p[i] ← i
  done;
  { current = p; new = true }

let next_permutation (c: cursor) : unit
= let p = c.current in
  let n = p.length in
  if n ≤ 1 then c.new ← false
  else
  let r = ref (n-2) in       (* 1. *)
  while !r ≥ 0 && p[!r] > p[!r+1] do
   invariant { -1 ≤ !r ≤ n-2 }
   variant   { !r + 1 }
   r := !r - 1
  done;
  if !r < 0 then (* last array reached. *)
   c.new ← false
  else                              (* 2. *)
   let j = ref (n-1) in
   while !j > !r + 1 && p[!r] ≥ p[!j] do
    invariant { !r + 1 ≤ !j ≤ n-1 }
    variant   { !j }
    j := !j - 1
   done;
   swap p !r !j;                    (* 3. *)
   reverse p (!r+1) n;              (* 4. *)
   c.new ← true
```

Listing 1 – Création du curseur et passage à la permutation suivante.

Génération de la structure suivante. Le contrat de la fonction `next` est

```
val next (c: cursor) : unit
  requires { sound c }
  ensures  { sound c }
  ensures  { c.current.length = (old c).current.length }
```

La précondition et la première postcondition spécifient la propriété de correction à l'aide du prédicat `sound`. La seconde postcondition assure que les structures d'entrée et de sortie ont la même taille.

Par exemple, une fonction `next_permutation` de passage à la permutation suivante est détaillée dans le listing 1. Elle utilise deux fonctions auxiliaires `swap` et `reverse` non reproduites ici. La fonction `swap` vient de la librairie standard de Why3. L'instruction `(swap` a i j`)` échange les éléments du tableau a aux indices i et j. La fonction `reverse` est telle que `(reverse` a l u`)` inverse la partie $a[l..u-1]$ du tableau a.

Afin de faciliter la lecture du code, les variables p et n représentent respectivement la permutation courante et sa taille. Si cette taille est 0 ou 1, la permutation courante est la dernière permutation (`c.new ← false`). Sinon, le programme procède par révision du suffixe du tableau p, comme détaillé dans l'exemple d'exécution suivant. Soit p le tableau d'entiers

i	0	1	2	3	4	5
$p[i]$	4	1	2	5	3	0

qui stocke les valeurs d'une permutation sur $[0..5]$, également notée p. Ainsi $p[i] = p(i)$ pour $i = 0, \ldots, 5$. Le programme transforme le tableau p en place, pour qu'il devienne le plus petit tableau p' strictement supérieur à p (selon l'ordre lexicographique $\prec$) qui représente une permutation p'. L'étape (1) cherche l'*indice de révision* $!r$ tel que p et p' aient le plus grand préfixe commun $p[0..!r-1] = p'[0..!r-1]$ (r est une référence et ! est l'opérateur de déréférencement). Si p est une permutation, cet indice est le plus grand indice i (le plus à droite) tel que $p[i]$ est inférieur à $p[i+1]$. Dans notre exemple de permutation p, l'indice de révision est $!r = 2$. Le *suffixe* est la fin du tableau $p[!r..5]$, à partir de l'indice de révision. L'étape (2) détermine la nouvelle valeur de $p[!r]$, telle que le tableau p' soit supérieur à p et le plus petit possible. Dans le cas d'une permutation, cette nouvelle valeur de $p[!r]$ est la plus petite valeur $p[!j]$ supérieure à $p[!r]$ et présente dans la partie $p[!r+1..5]$ du tableau après l'indice de révision. Dans notre exemple, c'est la valeur $p[4] = 3$, pour $!j = 4$. L'étape (3)

échange les valeurs de $p[!r]$ et $p[!j]$, grâce à la fonction `swap`. On obtient alors le tableau p_1 suivant :

$p_1[i]$	4	1	3	5	2	0

. L'étape (4) calcule la partie $p'[!r+1..5]$ la plus petite possible. Pour que p' soit une permutation, cette partie doit être la partie $p_1[!r+1..5]$ triée dans l'ordre croissant. Puisque cette partie $p_1[!r+1..5]$ est triée dans l'ordre décroissant, il suffit de l'inverser avec la fonction `reverse`, ce qui produit le tableau

$p'[i]$	4	1	3	0	2	5

. Si un indice de révision n'a pas été trouvé durant l'étape (1), alors $!r$ vaut -1 et p est la dernière permutation, ce qui est indiqué en attribuant la valeur `false` au champ `new` du curseur.

3 Propriétés

Correction. Dans la fonction `next_permutation`, les boucles `while` des étapes (1) et (2) sont spécifiées par un invariant qui fixe des bornes pour les indices de tableau modifiés par la boucle, et par un variant pour justifier leur terminaison. Avec ces annotations, la correction et la terminaison de cette fonction sont prouvées automatiquement avec Alt-Ergo 1.30.

Progression. Quand la fonction `next` de passage au suivant génère une nouvelle structure, elle doit satisfaire la propriété de progression

```
ensures { c.new → lt_lex (old c.current) c.current }
```

où le prédicat `lt_lex` formalise sur les tableaux d'entiers l'ordre lexicographique strict $\prec$ de la définition 1. Cet ordre est induit par l'ordre strict $<$ prédéfini sur les entiers du langage WhyML. Quand la structure courante est la dernière à générer, la fonction `next` ne doit pas la modifier, comme le stipule la postcondition

```
ensures { not c.new → array_eq (old c.current) c.current }
```

où le prédicat `array_eq` de la librairie standard de Why3 formalise l'égalité entre deux tableaux. Nous spécifions formellement l'ordre lexicographique grâce aux deux prédicats suivants :

```
predicate eq_prefix (a1 a2: array int) (u: int) = array_eq_sub a1 a2 0 u
predicate lt_lex (a1 a2: array int) = a1.length = a2.length ∧
  ∃ i:int. 0 ≤ i < a1.length ∧ eq_prefix a1 a2 i ∧ a1[i] < a2[i]
```

Le prédicat `eq_prefix` spécialise le prédicat `array_eq_sub` de la bibliothèque standard de Why3, qui est tel que `(array_eq_sub` a_1 a_2 l u`)` formalise l'égalité des sous-tableaux $a_1[l..u-1]$ et $a_2[l..u-1]$. Le prédicat `lt_lex` formalise la définition 1.

Après ajout de la postcondition

```
ensures { eq_prefix a (old a) l }
```

pour la fonction `reverse`, qui assure qu'elle ne modifie pas le préfixe $a[0..l-1]$ du tableau a, la preuve de la propriété de progression est automatique, avec Alt-Ergo 1.30.

Complétude. Dans cet article, nous réduisons la propriété de complétude aux trois propriétés suivantes : (1) La propriété *min_lex* impose que la première structure générée soit la plus petite structure selon l'ordre lexicographique. (2) La propriété *max_lex* exige que la dernière structure générée soit la plus grande selon l'ordre lexicographique. (3) La propriété d'*incrémentation* spécifie que la structure a_2 générée par la fonction `next` à partir de la structure a_1 est toujours la plus petite structure strictement supérieure à la structure a_1, selon l'ordre lexicographique. Autrement dit, il n'existe aucune structure a_3 tel que $a_1 < a_3 < a_2$.

Ces propriétés sont plus difficiles à démontrer que la correction et la progression, car elles incluent une quantification sur un tableau. Nous introduisons le lemme $(\forall a_1 a_2.\ a_1 \not\prec a_2 \rightarrow a_2 \preceq a_1)$ de totalité de l'ordre lexicographique. Ce lemme permet de valider automatiquement les propriétés *min_lex*

et *max_lex*. Cependant, les solveurs automatiques ne démontrent pas la propriété d'*incrémentation*. Nous envisageons de prouver interactivement sa traduction en Coq, produite par Why3 à partir du code WhyML. Nous commençons par le cas plus simple d'un générateur lexicographique de tableaux bornés (un tableau a est dit *borné* (par n) si $a[i] < n$ pour tous les indices i du tableau). Pour les tableaux bornés, les propriétés *min_lex* et *max_lex* sont également démontrées automatiquement, tandis que la propriété d'*incrémentation* l'est interactivement en Coq.

4 Conclusion

Nous avons présenté une formalisation générale des générateurs lexicographiques de structures combinatoires dans des tableaux d'entiers. Cette formalisation peut être généralisée à d'autres structures de données, comme les arbres. Nous avons spécifié et implémenté un générateur de permutations en WhyML, puis prouvé sa correction et sa progression avec Why3. Toutes les preuves sont effectuées sur une machine équipée d'un processeur Intel Core i3-2125 à 3.30 GHz $\times$ 4 sous Linux Ubuntu 16.04, en 1.2 secondes pour la fonction `next_permutation`, 0.3 secondes pour la fonction `reverse` et 0.06 secondes pour la fonction `create_cursor`.

Ce travail en cours doit se poursuivre avec une preuve de complétude pour le générateur de permutations, mais aussi avec la spécification, l'implémentation et la vérification déductive d'autres générateurs.

Références

[Alt18] The Alt-Ergo SMT solver. `http://alt-ergo.lri.fr`, 2018.

[BFM+18] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. *The Why3 Platform*, 2018. `http://why3.lri.fr/manual.pdf`.

[CDG12] M. Carlier, C. Dubois, and A. Gotlieb. A certified constraint solver over finite domains. In *FM'12*, volume 7436 of *LNCS*, pages 116–131. Springer, 2012.

[DG18] C. Dubois and A. Giorgetti. Test and proofs for custom data generators. 2018. À paraître.

[FP16a] J.-C. Filliâtre and M. Pereira. Itérer avec confiance. In *JFLA'16*, 2016. `https://hal.inria.fr/hal-01240891`.

[FP16b] J.-C. Filliâtre and M. Pereira. A modular way to reason about iteration. In *NFM'16*, volume 9690 of *LNCS*, pages 322–336. Springer, 2016.

[GGP15] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In *TAP'15*, volume 9154 of *LNCS*, pages 109–128. Springer, 2015.

[JD96] D. Jackson and C. Damon. Elements of style : Analyzing a software design feature with a counterexample detector. *IEEE Transactions on Software Engineering*, 22(7) :484–495, 1996.

[PHD+15] Z. Paraskevopoulou, C. Hriţcu, M. Dénès, L. Lampropoulos, and B. C. Pierce. Foundational property-based testing. In *ITP'15*, volume 9236 of *LNCS*, pages 325–343. Springer, 2015.

[SYC+04] K. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. *SIGSOFT Softw. Eng. Notes*, 29(4) :133–142, July 2004.