Gagnez sur tous les tableaux

Richard Genestier[1], Alain Giorgetti[1,2] et Guillaume Petiot[1,3]

1: Institut FEMTO-ST (UMR 6174), Université de Franche-Comté
16 route de Gray, 25030 Besançon CEDEX, France
richard.genestier,alain.giorgetti@femto-st.fr
2: Projet CASSIS, Inria, 54600 Villers-les-Nancy, France
3: CEA, LIST, Software Reliability Laboratory, PC 174, 91191 Gif-sur-Yvette France
guillaume.petiot@cea.fr

Résumé

Nous vérifions automatiquement des programmes impératifs d'énumération de structures combinatoires implantées dans des tableaux satisfaisant des propriétés structurelles données. Ces programmes C sont spécifiés en ACSL et vérifiés avec la plateforme Frama-C. La vérification déductive démontre automatiquement que tous les tableaux produits satisfont leurs propriétés structurelles. Elle est facilitée par sa combinaison avec des analyses dynamiques, qui permettent aussi de valider d'autres propriétés des programmes, comme leur exhaustivité. Nous proposons une bibliothèque de programmes vérifiés et des patrons de programmation et de spécification facilitant leur conception et leur mise au point. Ces programmes trouvent une application naturelle dans le test exhaustif borné de programmes manipulant ces tableaux.

1. Introduction

Ce travail se situe à la frontière entre deux domaines de recherche : la combinatoire énumérative et le génie logiciel. Il montre comment l'utilisation de méthodes et d'outils de vérification statique et dynamique facilite la conception et la mise au point de programmes d'énumération de structures combinatoires.

La combinatoire est la branche des mathématiques qui étudie les structures discrètes qui sont en nombre fini lorsque certains de leurs paramètres sont fixés. Au sein de cette discipline, la combinatoire énumérative cherche à compter le nombre de structures de même taille et à mettre au point des algorithmes efficaces pour les générer. Les permutations de n objets distinguables et les permutations particulières, telles que les involutions ou les dérangements, sont des exemples de structures combinatoires. La recherche en génie logiciel, quant à elle, étudie, développe et distribue des méthodes originales pour la conception rationnelle et la maintenance des logiciels. Dans cette discipline, le test logiciel et son automatisation ont une grande importance. Le test exhaustif borné (*bounded-exhaustive testing*, ou BET) [14] est une méthode élémentaire pour la génération automatique de données de test. Il est particulièrement adapté à la détection d'erreurs dans des programmes portant sur des données structurées, car il fournit des contre-exemples de taille minimale [13].

Ce travail est une contribution au test exhaustif borné de fonctions C dont une entrée est un tableau représentant une structure combinatoire. Par exemple, un tableau sans répétition de n entiers compris entre 0 et $n-1$ représente une permutation de ces n entiers. On dit alors que le tableau satisfait deux propriétés structurelles : la propriété que ses valeurs soient comprises entre 0 et $n-1$, qui lui donne la structure d'endofonction sur $\{0,\ldots,n-1\}$, et la propriété de non-répétition, qui lui donne la structure d'injection. Conjointement, ces deux propriétés donnent au tableau la structure de permutation. Pour énumérer ces structures, les combinatoriciens proposent des algorithmes efficaces [7]

et des catalogues de programmes qui les implantent, tels que le *fxtbook* [1]. Nous montrons ici comment les combinatoriciens peuvent tirer profit des méthodes de vérification du génie logiciel, pour concevoir et vérifier ces programmes.

Nous portons notre attention sur les algorithmes d'énumération combinatoire qui génèrent séquentiellement toutes les structures de même taille, selon un ordre total prédéfini. Nous désignons ici par *énumérateur* un tel algorithme, lorsqu'il est composé de deux fonctions : la première fonction construit la plus petite structure de taille donnée selon l'ordre prédéfini, la seconde fonction modifie une structure quelconque pour construire la structure suivante de même taille selon cet ordre. L'efficacité d'un énumérateur est liée à sa capacité à ne pas construire de structures intermédiaires qui ne satisfont pas les propriétés attendues. Nous étudions ici des implantations de ces énumérateurs en C, sur lesquelles nous effectuons deux sortes de vérifications : nous vérifions statiquement que les structures générées satisfont leurs propriétés structurelles, selon l'approche déductive promue par Hoare et Floyd. Ces démonstrations de correction (appelées "*vérifications*" dans toute la suite) sont complétées par des vérifications dynamiques (appelées "*validations*" dans toute la suite), entre autres pour s'assurer de l'exhaustivité des énumérateurs.

Pour la vérification statique, les programmes sont spécifiés formellement par des contrats ACSL. La vérification déductive est effectuée à l'aide de la plateforme de vérification de programmes Frama-C [6] et de son greffon WP, assisté par des solveurs SMT. En cas d'échec de la vérification statique, on utilise l'outil de vérification dynamique StaDy. Ce greffon de Frama-C permet de guider l'utilisateur dans son travail de preuve [11], en montrant des incohérences entre le code et la spécification (contre-exemples), si elles existent. En l'absence de contre-exemples, il fournit une garantie de conformité entre le code et sa spécification, par couverture par le test de tous les chemins faisables du code et de la spécification. Comme le nombre de chemins n'est fini que lorsqu'une valeur maximale est donnée pour la taille des structures, et que ce nombre de chemins augmente rapidement avec cette taille, il peut être jugé utile de généraliser cette garantie à toute taille par une vérification déductive.

Pour accélérer la production de nouveaux énumérateurs vérifiés, nous proposons plusieurs patrons de programmation et de spécification. Le premier permet de spécifier et vérifier rapidement un énumérateur qui procède par filtrage parmi des structures plus générales. Le deuxième généralise le principe d'énumération selon un ordre lexicographique, adopté par de nombreux algorithmes efficaces.

Les contributions de cet article sont :

— le codage en C et la spécification en ACSL d'algorithmes de génération de structures combinatoires,
— la preuve formelle automatique de la correction de ces programmes vis-à-vis de leur spécification formelle,
— des patrons généraux d'aide à l'écriture de ces programmes spécifiés, et
— une bibliothèque d'énumérateurs [1] vérifiés.

La partie 2 définit la notion d'énumérateur de structures combinatoires formellement spécifié, à travers un exemple fil-rouge. Les parties 3 et 4 proposent des patrons généraux pour ces énumérateurs, respectivement par filtrage dans une famille de structures plus vaste, et selon un ordre lexicographique. Les énumérateurs produits à l'aide de ces patrons sont présentés dans la partie 5. Les perspectives ouvertes par ce travail sont discutées dans la partie 6.

2. Principes et exemple

Nous présentons ici les principes de génération et de vérification de structures combinatoires mis en œuvre au fil de l'article. Après avoir présenté dans la partie 2.1 les différents langages et outils utilisés, nous définissons dans la partie 2.2 la notion d'énumérateur de structures. Enfin, nous introduisons dans la partie 2.3 un exemple fil-rouge qui servira à expliquer les formalismes utilisés et leur sémantique.

1. Archives `enum.*.tar.gz` téléchargeables depuis la page `http://members.femto-st.fr/alain-giorgetti/en`.

2.1. Langages et outils

Tous les algorithmes d'énumération étudiés ici sont programmés en C et spécifiés en ANSI C Specification Language (ACSL) [2]. ACSL est un langage dédié à l'analyse statique de programmes C. ACSL permet de spécifier formellement des contrats qui doivent être vérifiés par le programme, comme détaillé dans la partie 2.3. Nous utilisons la plateforme d'analyse de programmes C Frama-C, développée par le CEA LIST et INRIA Saclay.

Frama-C utilise le langage de spécification ACSL. Pour la vérification déductive, nous utilisons le greffon WP, qui implante le calcul de plus faible précondition (*Weakest Precondition Calculus*) pour des programmes C annotés en ACSL, et les solveurs SMT Alt-Ergo et CVC3. Ces solveurs sont utilisés par WP pour déterminer si une formule du premier ordre est satisfaisable. Pour faciliter la vérification déductive, nous utilisons le greffon StaDy [11, 10]. Ce greffon de Frama-C inspecte tous les chemins possibles de l'exécution du code du programme et de sa spécification. Dans les cas où l'explosion du nombre de chemins est problématique, il est possible de renforcer la précondition de la fonction pour le test, sans que cela n'affecte la précondition utilisée par la vérification déductive. Cela permet d'utiliser la vérification dynamique sur de petites instances du programme, en utilisant des tableaux de petite taille par exemple. Les contre-exemples trouvés sur ces instances sont tout aussi pertinents, et permettent de déceler rapidement des incohérences entre le code et les annotations de spécification. Il est possible de relâcher progressivement cette précondition pour le test, et ainsi d'obtenir, en l'absence de contre-exemples, une confiance de plus en plus grande dans la conformité du code vis à vis de sa spécification. Une fois les éventuels contre-exemples détectés et corrigés, l'utilisateur peut s'attendre à obtenir de meilleurs résultats concernant la vérification de son programme, et il relancera donc les solveurs automatiques par l'intermédiaire de WP.

2.2. Fonctions d'énumération

Nous étudions les algorithmes d'énumération qui génèrent séquentiellement des tableaux de même taille, selon un ordre lexicographique induit par un ordre sur les éléments des tableaux. Un tel ordre peut être 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 (induit par $<$ sur les tableaux)*, la relation binaire, également notée $<$, telle que, pour tout entier $n \geq 0$ et tous les tableaux b et c de taille n dont les éléments sont dans A, on a $b < c$ si et seulement s'il existe un indice i ($0 \leq i \leq n-1$) tel que $b[i] < c[i]$ et $b[j] = c[j]$ pour tout j entre 0 et $i-1$ inclus.*

La relation binaire $<$ ainsi définie est un ordre strict, qui est total si l'ordre sur A l'est. Dans toute la suite, A est l'ensemble des entiers relatifs (ou des entiers représentables de type `int` en C, puisqu'on ignore ici les problèmes de dépassement de capacité arithmétique), muni de son ordre strict total naturel, tel que $i < i+1$ pour tout entier i.

Considérons une structure combinatoire `YYY` encodée dans un tableau C `a` de taille `n` dont les éléments sont de type `XXXX`. Un *énumérateur* (en C) d'une telle structure est constitué de deux fonctions C, appelées *fonctions d'énumération* :

— La fonction `int firstYYY(XXXX a[], int n,...)` génère la première structure `YYY` de taille `n` dans le tableau `a` et retourne `1`, s'il existe au moins une structure de cette taille. Sinon, cette fonction retourne `0`.

— La fonction `int nextYYY(XXXX a[], int n, ...)` retourne `1` et génère dans le tableau `a` de taille `n` la structure `YYY` qui suit immédiatement celle qui est stockée dans le tableau `a` à l'appel de la fonction, si cette structure n'est pas la dernière. Sinon, elle retourne `0` et le contenu du tableau est inchangé.

Dans le profil de ces deux fonctions C, les pointillés représentent d'autres paramètres éventuellement

requis pour la génération de la structure combinatoire. On se limite ici aux cas où aucun de ces paramètres supplémentaires n'est une structure.

Un usage typique de ces deux fonctions, pour générer et utiliser toutes les structures de taille comprise entre `minsize` et `maxsize`, est :

```
for (n = minsize; n ≤ maxsize; n++) {
 firstYYY(s,n);
 // Treatment of the first structure

 while (nextYYY(s,n) == 1) {
  // Treatment of the next structure
 }
}
```

où `s` est un tableau C de taille suffisante, dont les éléments sont de type `XXXX`. Tant que la dernière structure n'est pas atteinte, on génère la suivante.

2.3. Exemple

En guise de fil rouge, considérons l'exemple des fonctions à croissance limitée, définies comme suit :

Définition 2. *[1, page 235] Une* fonction à croissance limitée *(RGF, for* Restricted Growth Function*) de taille n est une endofonction r de $\{0,\dots,n-1\}$ telle que $r(0) = 0$ et $r(k) \leq r(k-1)+1$ pour $1 \leq k \leq n-1$.*

On peut représenter une endofonction r de $\{0,\dots,n-1\}$, et donc une RGF, par le tableau C de ses images :

0	1	…	$n-1$
$r(0)$	$r(1)$	…	$r(n-1)$

Ce tableau est de taille n et ses valeurs sont des entiers de $\{0,\dots,n-1\}$. Le *fxtbook* propose un algorithme efficace [1, page 235] pour calculer la RGF qui suit immédiatement une RGF r donnée, dans l'ordre lexicographique croissant :

1. Trouver l'entier j maximal tel que $r[j] \leq r[j-1]$.
2. Si un tel entier existe, incrémenter la valeur $r[j]$ et fixer $r[i] = 0$ pour tout $i > j$. Les autres valeurs de r restent inchangées.
3. Sinon, la génération est terminée, r était la plus grande RGF.

Par exemple, selon cet ordre, les cinq RGFs de taille 3 sont 000, 001, 010, 011 et 012 (dans cette notation, le tableau r est représenté par le mot $r[0]r[1]\cdots r[n-1]$ de ses valeurs). Toujours selon cet ordre, la première RGF est la fonction constante égale à 0.

Le code C et la spécification ACSL des fonctions `firstRGF` et `nextRGF` sont donnés dans le listing 1. Nous expliquons les aspects du langage de spécification ACSL que nous utilisons à travers l'exemple de la fonction `nextRGF`. Les annotations ACSL de la fonction `firstRGF` sont analogues. Pour faciliter la lecture des spécifications, certaines notations ACSL sont remplacées par des symboles mathématiques (par exemple, les mots-clés `\forall` et `integer` sont respectivement notés ∀ et ℤ).

La propriété caractéristique des RGFs selon la définition 2 est exprimée en ACSL par le *prédicat* `is_RGF`, défini comme suit, à partir de deux prédicats intermédiaires `is_non_neg` et `is_le_pred` :

```
1 /*@ predicate is_non_neg(int *a, ℤ b, ℤ c) = (∀ ℤ i; b ≤ i < c ⇒ a[i] ≥ 0);
2   @ predicate is_le_pred(int *a, ℤ b, ℤ c) = (∀ ℤ i; b ≤ i < c ⇒ a[i] ≤ a[i-1]+1);
3   @ predicate is_RGF(int *a, ℤ n) = a[0] == 0 ∧ is_non_neg(a,0,n) ∧ is_le_pred(a,1,n); */
```

Ces prédicats sont définis dans le fichier source C avant le code des fonctions `firstRGF` et `nextRGF`. Ils sont définis avec un paramètre supplémentaire utile pour spécifier les invariants de boucle. De plus, le prédicat supplémentaire `is_zero` (défini dans les lignes 1 et 2 du listing 1) est introduit pour exprimer l'invariant de la boucle de la fonction `firstRGF`.

```
1  /*@ predicate is_zero(int *a, ℤ b, ℤ c) =
2    @  (∀ ℤ i; b ≤ i < c ⇒ a[i] == 0); */
3
4  /*@ requires n > 0 ∧ \valid(r+(0..n-1));
5    @ assigns r[0..n-1];
6    @ ensures is_RGF(r,n); */
7  void firstRGF(int r[], int n) {
8   int k;
9   /*@ loop invariant 0 ≤ k ≤ n;
10    @ loop invariant is_zero(r,0,k);
11    @ loop assigns k, r[0..n-1];
12    @ loop variant n-k; */
13  for (k = 0; k < n; k++) { r[k] = 0; }
14 }
15
16 /*@ requires n > 0 ∧ \valid(r+(0..n-1));
17   @ requires is_RGF(r,n);
18   @ assigns r[1..n-1];
19   @ ensures is_RGF(r,n); */
20 int nextRGF(int r[], int n) {
21  int i,k;
22  /*@ loop invariant 0 ≤ i ≤ n-1;
23    @ loop assigns i;
24    @ loop variant i-1; */
25  for (i = n-1; i ≥ 1; i--)
26   if (r[i] ≤ r[i-1]) { break; }
27  if (i == 0) { return 0; } // Last RGF.
28  r[i] = r[i] + 1;
29  /*@ loop invariant i+1 ≤ k ≤ n;
30    @ loop invariant is_non_neg(r,0,k);
31    @ loop invariant is_le_pred(r,1,k);
32    @ loop assigns k, r[i+1..n-1];
33    @ loop variant n-k; */
34  for (k = i+1; k < n; k++) { r[k] = 0; }
35  return 1;
36 }
```

Listing 1 – Génération efficace des RGFs en C/ACSL.

Avant l'entête d'une fonction, une annotation `requires R;` spécifie que la *précondition* `R` doit être vérifiée par les paramètres de la fonction lorsqu'elle est appelée. Ainsi, on exige dans la ligne 16 que le tableau `r` soit de taille `n` strictement positive et soit alloué en mémoire. Dans la ligne 17 on exige que le tableau `r` représente une RGF. Une annotation de la forme `assigns A;` déclare dans `A` les paramètres de la fonction qui peuvent être modifiés lors de son exécution. Ainsi, la ligne 18 déclare que tous les éléments du tableau `r` peuvent être modifiés, sauf le premier `r[0]`. Une annotation `ensures E;` affirme que la *postcondition* `E` est vraie en sortie de fonction. Dans la ligne 19, on affirme que le tableau `r` sera une RGF après exécution de la fonction.

Dans le corps des fonctions, on utilise les annotations ACSL suivantes. Une annotation `loop invariant I;` immédiatement avant une boucle déclare que la formule `I` est un *invariant* de cette boucle, c'est-à-dire une propriété qui doit être établie et préservée à chaque passage dans la boucle. Par exemple, avant la deuxième boucle de la fonction `nextRGF`, trois invariants de boucle affirment successivement que la variable de boucle `k` reste entre `i+1` et `n` (ligne 29), que les `k` premiers éléments du tableau sont positifs ou nuls (ligne 30), et que la propriété `is_le_pred` est satisfaite jusqu'à `k` (ligne 31). L'annotation `loop assigns` définit un sur-ensemble des variables dont la valeur peut être modifiée par une itération de la boucle. Par exemple `loop assigns x, t1[i], t2[a..b], t3[..]` signifie que les seules valeurs qui peuvent changer sont celles de `x`, de `t1[i]`, des éléments de `t2` entre les indices `a` et `b`, et de tous les éléments du tableau `t3`. En revanche cela ne signifie pas que toutes ces valeurs vont certainement être modifiées, et encore moins qu'elles seront modifiées à chaque itération. Mais l'annotation doit tenir compte de toutes les itérations possibles de la boucle. L'annotation `loop variant V;` définit un *variant de boucle* `V` qui peut être utilisé pour garantir la terminaison de la boucle. Cette expression entière doit rester positive ou nulle et décroître strictement entre deux itérations successives de la boucle. Par exemple, l'expression `n-k` est un variant de la boucle de la ligne 34 du listing 1.

2.4. Vérification

Supposons que le listing 1 soit le contenu d'un fichier `rgf.c`. La vérification statique de la fonction `nextRGF` avec Frama-C assisté de WP s'effectue par exécution de la commande `frama-c -wp-fct nextRGF rgf.c`. Frama-C indique si chaque obligation de preuve générée par WP est prouvée par le solveur SMT Alt-Ergo, en indiquant la durée de chaque preuve.

3. Génération par filtrage

Les structures combinatoires qu'on cherche à générer sont très souvent des structures classiques d'une certaine nature, définies par une propriété caractéristique appelée leur *invariant*. Pour générer les structures classiques (telles que les n-uplets, les permutations, ou les combinaisons de k éléments parmi n), la littérature en combinatoire énumérative nous propose des algorithmes efficaces. La génération *par filtrage*, qui consiste à sélectionner parmi des structures celles qui satisfont une propriété donnée, fournit rapidement un premier énumérateur. Plus le filtrage élimine de structures, moins cet enumérateur est efficace. Cependant, il peut être suffisant pour détecter une erreur dans un programme ou pour valider une conjecture portant sur ces structures.

La partie 3.1 illustre le principe de génération par filtrage sur l'exemple des RGFs. La partie 3.2 formalise ce principe, en proposant un patron commun pour tous les énumérateurs par filtrage. Ce patron est à la fois un code C et une spécification ACSL génériques, dont les "trous" sont destinés à être remplacés par des noms et des invariants concrets. Pour instancier ce patron, il est nécessaire d'implanter l'invariant des sous-structures, sous la forme d'une fonction booléenne. La partie 3.3 propose un patron général pour cette fonction booléenne et sa spécification. Ces patrons eux-mêmes ont été vérifiés automatiquement.

3.1. Exemple

La famille des RGFs est une sous-famille de la famille des endofonctions de $\{0, ..., n-1\}$. Supposons que nous ayons déjà implanté, spécifié et vérifié automatiquement un énumérateur des endofonctions de $\{0, ..., n-1\}$, constitué des deux fonctions d'énumération `firstEndofct(a,n)` et `nextEndofct(a,n)`. Nous utilisons cet énumérateur pour implanter très rapidement un énumérateur des RGFs, par filtrage des endofonctions de $\{0, ..., n-1\}$ qui sont des RGFs. Dans cet exemple, le corps de la fonction `firstRGF(a,n)` est réduit à l'instruction `return firstEndofct(a,n);`, car la première endofonction de $\{0, ..., n-1\}$ générée dans le tableau `a` est une RGF. Dans le cas général, la fonction d'énumération de la première structure doit implanter le filtrage, comme détaillé dans la partie 3.2.

Le listing 2 propose une implantation de la fonction `nextRGF(a,n)` qui énumère les endofonctions de $\{0, ..., n-1\}$ à partir de `a`, et qui ignorent celles qui ne satisfont pas les conditions de la définition 2. Dans ce but, cette fonction appelle la fonction C `isRGF`, qui caractérise une RGF parmi les endofonctions de $\{0, ..., n-1\}$. Cette fonction est définie de la ligne 8 à la ligne 26 du listing 2. Sa postcondition ligne 14 déclare que son tableau d'entrée satisfait le prédicat `is_RGF` lorsqu'elle retourne 1. Nous verrons dans la partie 3.3 comment on peut systématiser l'écriture de cette fonction à partir du prédicat correspondant.

```
1  /*@ predicate is_le_pred(int *a, ℤ n) =
2    @  (∀ ℤ i; 1 ≤ i < n ⇒ a[i] ≤ a[i-1]+1); */
3
4  /*@ predicate is_RGF(int *a, ℤ n) =
5    @  is_endofct(a,n) ∧ a[0] == 0
6    @  ∧ is_le_pred(a,n); */
7
8  /* Returns 1 when the array a is an RGF, and
9   * 0 otherwise. */
10 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
11   @ requires is_endofct(a,n);
12   @ assigns \nothing;
13   @ ensures \result == 0 ∨ \result == 1;
14   @ ensures \result == 1 ⇔ is_RGF(a,n); */
15 int isRGF(int a[], int n) {
16  int i;
17
18  if (a[0] ≠ 0) return 0;
19  /*@ loop invariant 1 ≤ i ≤ n;
20    @ loop invariant is_le_pred(a,i);
21    @ loop assigns i;
22    @ loop variant n-i; */
23  for (i = 1; i < n; i++)
24    if (a[i] > a[i-1]+1) return 0;
25  return 1;
26 }
27
28 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
29   @ requires is_RGF(a,n);
30   @ assigns a[0..n-1];
31   @ ensures \result == 0 ∨ \result == 1;
32   @ ensures \result == 1 ⇒ is_RGF(a,n); */
33 int nextRGF(int a[], int n) {
34  int tmp;
35
36  /*@ loop invariant is_endofct(a,n);
37    @ loop assigns a[0..n-1],tmp; */
38  do {
39   tmp = nextEndofct(a,n);
40  } while (tmp ≠ 0 ∧ isRGF(a,n) == 0);
41  if (tmp == 0) { return 0; }
42  return 1;
43 }
```

Listing 2 – RGF suivante par filtrage.

3.2. Patron général de génération par filtrage

La génération des RGFs par filtrage se généralise à de nombreuses familles de structures combinatoires, définies par des propriétés supplémentaires de structures plus générales. Le listing 3 propose un patron général pour la génération des structures `ZZZ` par filtrage des structures `XXX` qui ne satisfont pas l'invariant `is_YYY`, implanté par la fonction booléenne `isYYY`. Les structures `XXX` sont représentées par les tableaux de `n` entiers qui satisfont la propriété `is_XXX`.

```
1  /*@ axiomatic preds {
2    @  predicate is_XXX(int *a, ℤ n)
3    @    reads a[0..n-1];
4    @  predicate is_YYY(int *a, ℤ n)
5    @    reads a[0..n-1];
6    @  predicate is_ZZZ(int *a, ℤ n) =
7    @    is_XXX(a,n) ∧ is_YYY(a,n);
8    @ } */
9
10 /* Returns 1 when the array a satisfies
11  * property is_YYY, and 0 otherwise. */
12 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
13   @ assigns \nothing;
14   @ ensures \result == 0 ∨ \result == 1;
15   @ ensures \result == 1 ⇔ is_YYY(a,n); */
16 int isYYY(int a[], int n);
17
18 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
19   @ assigns a[0..n-1];
20   @ ensures \result == 0 ∨ \result == 1;
21   @ ensures \result == 1 ⇒ is_XXX(a,n); */
22 int firstXXX(int a[], int n);
23
24 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
25   @ requires is_XXX(a,n);
26   @ assigns a[0..n-1];
27   @ ensures \result == 0 ∨ \result == 1;
28   @ ensures \result == 1 ⇒ is_XXX(a,n); */
29 int nextXXX(int a[], int n);
30
31 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
32   @ assigns a[0..n-1];
33   @ ensures \result == 0 ∨ \result == 1;
34   @ ensures \result == 1 ⇒ is_ZZZ(a,n); */
35 int firstZZZ(int a[], int n) {
36  int tmp;
37
38  tmp = firstXXX(a,n);
39  /*@ loop invariant tmp ≠ 0 ⇒ is_XXX(a,n);
40    @ loop assigns a[0..n-1],tmp; */
41  while (tmp ≠ 0 ∧ isYYY(a,n) == 0) {
42   tmp = nextXXX(a,n);
43  }
44  if (tmp == 0) { return 0; }
45  return 1;
46 }
47
48 /*@ requires n > 0 ∧ \valid(a+(0..n-1));
49   @ requires is_ZZZ(a,n);
50   @ assigns a[0..n-1];
51   @ ensures \result == 0 ∨ \result == 1;
52   @ ensures \result == 1 ⇒ is_ZZZ(a,n); */
53 int nextZZZ(int a[], int n) {
54  int tmp;
55
56  /*@ loop invariant is_XXX(a,n);
57    @ loop assigns a[0..n-1],tmp; */
58  do {
59   tmp = nextXXX(a,n);
60  } while (tmp ≠ 0 ∧ isYYY(a,n) == 0);
61  if (tmp == 0) { return 0; }
62  return 1;
63 }
```

Listing 3 – Patron de la génération par filtrage.

Dans ce patron, les expressions des prédicats `is_XXX` et `is_YYY` sont quelconques. C'est pourquoi elles sont déclarées en ACSL dans un bloc axiomatique, de la ligne 1 à la ligne 8 du listing 3. En supposant que les fonctions `isYYY`, `firstXXX` et `nextXXX` satisfont leurs spécifications, la correction des fonctions `firstZZZ` et `nextZZZ` est prouvée automatiquement avec Frama-C et WP assistés d'Alt-Ergo, comme détaillé dans la partie 5.

L'énumérateur des RGFs par filtrage (listing 2) s'obtient en instanciant le patron général comme suit : on remplace respectivement `XXX`, `YYY` et `ZZZ`, par `Endofct`, `RGF` et `RGF`, et on implante la propriété `is_RGF` sous forme de fonction booléenne. D'autres exemples d'utilisation de ce patron sont donnés dans la partie 5.

Ainsi, à partir d'un énumérateur d'une famille de structures combinatoires spécifié et vérifié, on obtient rapidement des énumérateurs spécifiés de leurs sous-familles, le processus pouvant s'itérer. Leur vérification automatique dépend essentiellement de celle de la fonction booléenne qui implante leur invariant.

3.3. Patron général des fonctions booléennes

Nous proposons à présent un patron pour le contrat ACSL et le code C d'une fonction booléenne correspondant à une propriété structurelle d'un tableau exprimée en logique du premier ordre. Si la propriété n'est qu'une combinaison booléenne de prédicats atomiques, la correspondance est évidente :

les opérateurs booléens (comme la conjonction ou la négation) soit existent en C, soit peuvent être facilement exprimés par une combinaison d'opérateurs C. Ainsi, les cas intéressants sont les formules avec quantificateurs. Le cas d'un seul quantificateur est trop restrictif. Le cas général, avec des quantificateurs imbriqués et combinés avec des opérateurs booléens de manière arbitraire, serait trop lourd à formaliser. Le résultat serait peu lisible. Nous avons choisi de présenter le cas de deux quantificateurs imbriqués. Il suffit pour donner une idée de ce que serait le cas général, et il est utile en lui-même.

Les listings 4 et 5 proposent un patron qui couvre tous les cas où la propriété est de la forme $\forall i.\, 0 \leq i < n \Rightarrow (\exists j.\, 0 \leq j < n \wedge \varphi)$, où φ est une formule sans quantificateurs exprimant une propriété dépendant de i et de j et portant sur un tableau de taille n. Cette propriété est décomposée en trois prédicats `is_X1(a,n)`, `is_X2(a,n,v1)` et `is_X3(a,n,v1,v2)`, correspondant respectivement à la propriété universelle complète, à la sous-formule existentielle et à la propriété φ qu'elle quantifie, pour un tableau `a` de taille `n`. Le paramètre supplémentaire `v1` du prédicat `is_X2` correspond à la variable libre i des sous-formules $(\exists j.\, 0 \leq j < n \wedge \varphi)$ et φ. De même, le paramètre supplémentaire `v2` du prédicat `is_X3` correspond à la variable libre j de φ. Cette formule sans quantificateurs φ étant quelconque, le prédicat correspondant `is_X3` n'est pas défini, mais seulement déclaré dans un bloc axiomatique, dans les lignes 1 à 3 du listing 4.

```
1 /*@ axiomatic preds {
2   @  predicate is_X3(int *a, ℤ n, ℤ v1, ℤ v2) reads a[0..n-1];
3   @ }
4   @ predicate is_X2_gen(int *a, ℤ n, ℤ v1, ℤ v2) = ∃ ℤ i2; 0 ≤ i2 < v2 ∧ is_X3(a,n,v1,i2);
5   @ predicate is_X2(int *a, ℤ n, ℤ v1) = is_X2_gen(a,n,v1,n);
6   @ predicate is_X1_gen(int *a, ℤ n, ℤ v1) = ∀ ℤ i1; 0 ≤ i1 < v1 ⇒ is_X2(a,n,i1);
7   @ predicate is_X1(int *a, ℤ n) = is_X1_gen(a,n,n); */
```

Listing 4 – Prédicats d'une propriété de la forme $\forall\exists$.

```
1  /*@ requires n ≥ 0 ∧ \valid(a+(0..n-1));
2    @ assigns \nothing;
3    @ ensures \result == 0 ∨ \result == 1;
4    @ ensures \result == 1 ⇔ is_X3(a,n,v1,v2); */
5  int isX3(int a[], int n, int v1, int v2);
6
7  /*@ requires n ≥ 0 ∧ \valid(a+(0..n-1));
8    @ assigns \nothing;
9    @ ensures \result == 0 ∨ \result == 1;
10   @ ensures \result == 1 ⇔ is_X2(a,n,v1); */
11 int isX2(int a[], int n, int v1) {
12  int i;
13
14  /*@ loop invariant 0 ≤ i ≤ n;
15    @ loop invariant ! is_X2_gen(a,n,v1,i);
16    @ loop assigns i;
17    @ loop variant n-i; */
18  for (i = 0; i < n; i++)
19   if (isX3(a,n,v1,i) == 1) return 1;
20  return 0;
21 }
```

```
22
23 /*@ requires n ≥ 0 ∧ \valid(a+(0..n-1));
24   @ assigns \nothing;
25   @ ensures \result == 0 ∨ \result == 1;
26   @ ensures \result == 1 ⇔ is_X1(a,n); */
27 int isX1(int a[], int n) {
28  int i;
29
30  /*@ loop invariant 0 ≤ i ≤ n;
31    @ loop invariant is_X1_gen(a,n,i);
32    @ loop assigns i;
33    @ loop variant n-i; */
34  for (i = 0; i < n; i++)
35   if (isX2(a,n,i) == 0) return 0;
36  return 1;
37 }
```

Listing 5 – Fonction booléenne associée à une propriété de la forme $\forall\exists$.

On dit que la fonction booléenne `isX1` *implante* le prédicat `is_X1` pour exprimer qu'elle retourne `1` si ses paramètres satisfont le prédicat, et `0` sinon. De même, les fonctions booléennes `isX2` et `isX3` implantent respectivement les prédicats `is_X2` et `is_X3`. Pour $k = 1, 2$, la fonction `isX`k implante une boucle qui évalue successivement le prédicat `is_X`$(k+1)$ pour tous les éléments du tableau. Puisque la fonction booléenne `is_X1` implante une propriété universelle, elle retourne `0` dès qu'un élément qui ne satisfait pas la formule quantifiée est rencontré. Sinon, elle retourne `1`. L'invariant de boucle spécifie que la propriété `is_X2` est vraie jusqu'à l'indice courant `i` de parcours du tableau. Dualement, puisque la fonction booléenne `is_X2` implante une propriété existentielle, elle retourne `1` dès qu'un élément satisfaisant la formule quantifiée est rencontré. Sinon, elle retourne `0`. L'invariant de boucle spécifie que la propriété `is_X2` est fausse jusqu'à l'indice courant `i` de parcours du tableau. Ces invariants de

boucle sont spécifiés à l'aide d'une généralisation du prédicat `is_X`k, nommée `is_X`k`_gen`, définie dans les lignes 4 et 6 du listing 4.

Supposons que les listings 4 et 5 forment dans cet ordre le contenu d'un fichier `allex.c`. Admettons que la fonction booléenne `isX3` est conforme à sa spécification. Par la commande

```
frama-c -wp allex.c -wp-skip-fct isX3
```

on prouve alors automatiquement que les autres fonctions satisfont leur spécification.

Une application immédiate du patron précédent est la génération des endosurjections par filtrage des endofonctions. On appelle *endosurjection de* $\{0, ..., n-1\}$ toute endofonction de $\{0, ..., n-1\}$ qui est une surjection. Une endosurjection f satisfait la propriété $\forall i.\, 0 \leq i < n \Rightarrow (\exists j.\, 0 \leq j < n \wedge f(j) = i)$. Un énumérateur des endosurjections s'obtient simplement en complétant les listings 4 et 5 avec le listing 3 du patron de génération par filtrage, en renommant `X1`, `X2`, `XXX`, `YYY` et `ZZZ` respectivement en `im`, `eq_im`, `endofct`, `im` et `surj`, en définissant le prédicat `is_X3` par

```
predicate is_X3(int *a, ℤ n, ℤ v1, ℤ v2) = a[v2] == v1;
```

et en implantant la fonction `isX3` avec l'instruction unique `return(a[v2] ==v1);`. A partir de l'énumérateur d'endofonctions déjà utilisé pour les RGFs dans la partie 3.1, la mise au point et la vérification automatique de cet énumérateur d'endosurjections s'effectuent en quelques minutes. Après ce travail minimal, il est possible d'apporter diverses simplifications à cet énumérateur, tout en préservant sa vérification automatique. Par exemple, on peut supprimer le premier paramètre du prédicat `is_X3`, qui n'est pas utilisé dans cet exemple.

Pour le second cas d'alternance de deux quantificateurs, c'est-à-dire pour les propriétés de la forme $\exists i.\, 0 \leq i < n \wedge (\forall j.\, 0 \leq j < n \Rightarrow \varphi)$, où φ est une propriété sans quantificateurs portant sur un tableau de taille n, nous proposons un patron nommé exall.

4. Génération dans un ordre lexicographique

Un énumérateur obtenu par filtrage est rarement optimal. Quand le filtrage est très fréquent, ses performances peuvent être très mauvaises. Lorsqu'il est nécessaire de générer des structures de manière plus efficace, on peut implanter des algorithmes connus en énumération combinatoire. Selon [12], "*For many combinatorial objects, the fastest known algorithms for listing [...] are with respect to lexicographic order.*". Nos connaissances en combinatoire énumérative ne nous permettent pas d'apprécier la pertinence de cette affirmation très générale. Nous avons temporairement choisi d'y adhérer, et nous avons implanté divers algorithmes de génération dans un ordre lexicographique. Les fonctions `next...` de ces programmes suivent toutes le même principe, appelé ici "*révision du suffixe*", que nous décrivons sur un exemple, avant de le généraliser en proposant un patron de code C et d'annotations ACSL pour cette fonction d'énumération.

Reprenons l'exemple des RGFs du listing 1. Dans l'explication suivante, on identifie le tableau `r` avec le mot `r[0] r[1]` $\cdots$ `r[n-1]`. La fonction `nextRGF` est constituée de trois parties :

— Dans les lignes 25 et 26, une boucle parcourt le mot de droite à gauche, jusqu'à trouver une position à partir de laquelle le suffixe du mot sera modifié. Cette position est appelée *indice de révision* du tableau `r`. Dans cet exemple, l'indice de révision est atteint lorsqu'on rencontre l'élément le plus à droite (i.e. d'indice maximal) inférieur ou égal à son prédécesseur.

— Si cette recherche échoue, alors la dernière structure est atteinte (ligne 27).

— Sinon, le contenu du tableau est modifié, de l'indice de révision à la fin du tableau, afin que le nouveau tableau soit le successeur du tableau courant selon l'ordre lexicographique, ayant le même préfixe jusqu'à l'indice de révision exclu. La manière de faire cette révision est dictée par les propriétés que le tableau doit respecter. Pour les RGFs, la propriété `r[i]` $\leq$ `r[i-1]` de l'indice de révision `i` permet d'incrémenter `r[i]` (ligne 28) et de remplir le reste du tableau avec `0` (ligne 34), pour obtenir le plus petit tableau qui satisfait la propriété de croissance limitée et

qui a le même contenu que `r` du début du tableau jusqu'à l'indice de révision exclu.

Le listing 6 présente le patron d'une fonction `nextXXX` d'un énumérateur qui suit ce principe de révision du suffixe. En remplaçant dans ce patron les pointillés par du code adapté, on produit plus rapidement un énumérateur dans l'ordre lexicographique. Un énumérateur par filtrage, rapidement mis au point au préalable, pourra servir d'implantation de référence pour valider cet algorithme optimisé, comme détaillé dans la partie 5.

```
1  /*@ requires n > 0 ∧ \valid(a+(0..n-1));
2    @ requires is_ZZZ(a,n,...);
3    @ assigns a[0..n-1];
4    @ ensures is_ZZZ(a,n,...); */
5  int nextZZZ(int a[], int n, ...) {
6   int i,rev;
7   // Search of the revision index:
8   /*@ loop invariant -1 ≤ rev ≤ n-1;
9     @ loop assigns rev;
10    @ loop variant rev; */
11  for (rev = n-1; rev ≥ ...; rev--) {
12   if (...) { break; }
13  }
14  if (rev == ...) { // Last structure reached.
15   return 0;
16  }
17  // Suffix revision:
18  /*@ assigns a[rev]; */
19  a[rev] = ...;
20  /*@ loop invariant rev+1 ≤ i ≤ n;
21    @ loop invariant ...;
22    @ loop assigns i,a[rev+1..n-1];
23    @ loop variant n-i; */
24  for (i = rev+1; i < n; i++) {
25    a[i] = ...;
26  }
27  return 1;
28 }
```

Listing 6 – Patron du successeur par révision du suffixe.

5. Expérimentations

Notre campagne d'expérimentations a pour objectif la mise en œuvre de la méthode suggérée dans les parties précédentes, pour produire une bibliothèque d'énumérateurs vérifiés. La partie 5.1 présente tous les énumérateurs dont la correction a été prouvée automatiquement durant ces expérimentations. La partie 5.2 détaille les divers mécanismes mis en œuvre pour valider l'exhaustivité des énumérateurs.

5.1. Enumérateurs prouvés

Notre campagne d'expérimentations se décompose en trois temps. Dans un premier temps, nous implantons, spécifions et vérifions des algorithmes efficaces d'énumération séquentielle de structures combinatoires, issus de la littérature. Dans un deuxième temps, en utilisant les patrons de la partie 3, nous obtenons rapidement par filtrage des énumérateurs de sous-familles de ces familles de structures. Dans un troisième temps, dans la mesure du possible, nous implantons d'autres énumérateurs de ces sous-familles, plus efficaces, et nous utilisons les énumérateurs par filtrage pour les valider, comme détaillé dans la partie 5.2.

Les structures énumérées sont représentées par un tableau de taille strictement positive n, dont les éléments sont des entiers entre 0 et k. Cet entier k est égal à $n-1$ si la structure ne nécessite qu'un seul paramètre. Ces tableaux sont toujours générés dans l'ordre lexicographique induit par la relation de précédence stricte $<$ sur les entiers.

Les résultats expérimentaux sont collectés dans la table 1. La première colonne donne le nom de la famille de structures générées. Ces noms sont introduits dans la suite de cette partie. Le nombre de lignes de code, comptabilisé dans la deuxième colonne, ne prend pas en compte les lignes constituées uniquement d'accolades. Le nombre de lignes d'ACSL n'est pas donné car il est presque le même dans tous les programmes. La correction de ces programmes a été prouvée automatiquement avec Frama-C Neon 20140301 et son greffon WP 0.8 utilisant Why3 0.82 assisté des solveurs Alt-Ergo 0.95.2, CVC3 2.4.1 et CVC4 1.3. La troisième colonne indique le nombre d'obligations de preuves (buts) générés par WP, et la quatrième (resp. cinquième) colonne donne la durée de la preuve de ces buts par le solveur Alt-Ergo (resp. Alt-Ergo, CVC3 et CVC4), en secondes, sur un PC Intel Core i5-3230M à 2.60GHz × 4 sous Linux Ubuntu 12.04.

Exemple	nb. lignes code	nb. buts	durée Alt-Ergo (s)	durée Alt-Ergo + CVC3 + CVC4 (s)
filtering	14	47	0.452	0.496
allex	11	40	0.204	0.236
exall	12	40	0.224	0.286
fct	13	36	1.104	0.672
subset	13	35	0.628	0.548
endofct	4	13	0.140	0.162
rgf $\subset$ endofct	25	63	0.616	0.564
comb $\subset$ fct	21	63	0.508	0.612
inj $\subset$ fct	17	87	0.736	0.750
surj $\subset$ fct	29	99	0.840	0.886
perm $\subset$ fct	30	89	0.732	0.804
perm = endofct $\wedge$ inj	29	86	0.752	0.742
perm = endofct $\wedge$ surj	28	98	0.792	0.764
invol $\subset$ perm	20	62	0.560	0.624
derang $\subset$ perm	20	62	0.532	0.608
rgf	13	36	2.8	2.920
comb	20	87	Timeout	54.5852

TABLE 1 – Résultats de vérification.

Le premier bloc concerne les patrons présentés dans la partie 3. Dans le deuxième bloc, on désigne par fct un énumérateur efficace de la famille des fonctions de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$. Le troisième bloc concerne les spécialisations décrites dans la partie 5.1.1, le quatrième bloc concerne les générations par filtrage (partie 5.1.2) et le cinquième bloc concerne les algorithmes efficaces décrits dans la partie 5.1.3.

5.1.1. Spécialisation d'une famille

Lorsqu'une famille de structures combinatoires a d'autres paramètres que sa taille, on peut fixer la valeur de certains de ces paramètres et obtenir ainsi facilement d'autres énumérateurs. On dit qu'on a *spécialisé* la famille. Par exemple, la spécialisation de la famille des fonctions de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$ pour $k = 1$ donne les fonctions booléennes, qui codent la famille subset des sous-ensembles d'un ensemble à n éléments [1, page 203]). La spécialisation avec $k = n - 1$ donne la famille endofct des endofonctions de $\{0, ..., n-1\}$.

5.1.2. Filtrage à partir des fonctions

On désigne par zzz $\subset$ xxx (resp. zzz = xxx $\wedge$ yyy) un énumérateur des structures zzz par filtrage parmi les structures plus générales xxx (resp. des structures qui satisfont la propriété supplémentaire des structures yyy). Par exemple, rgf $\subset$ endofct désigne l'énumérateur des fonctions à croissance limitée par filtrage parmi les endofonctions (présenté dans la partie 3.2).

À partir de l'énumérateur fct des fonctions de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$, en utilisant les patrons présentés dans la partie 3, nous avons généré par filtrage :

— Les combinaisons de p éléments choisis parmi n (énumérateur comb $\subset$ fct) : la combinaison $\{e_0, \ldots, e_{p-1}\}$, avec $0 \leq e_0 < \ldots < e_{p-1} \leq n-1$ est représentée par la fonction c de $\{0, \ldots, p-1\}$ dans $\{0, ..., n-1\}$ telle que $c[i] = e_i$.

— Les injections de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$ $(n \leq k+1)$, énumérateur inj $\subset$ fct.

— Les surjections de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$ $(n \geq k+1)$, énumérateur surj $\subset$ fct.

En combinant spécialisation et filtrage, on peut produire quatre énumérateurs des permutations de $\{0, ..., n-1\}$ (structure perm) :

1. perm $\subset$ surj, par spécialisation à partir des surjections de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$ (pour $k = n-1$),
2. perm $\subset$ inj, par spécialisation à partir des injections de $\{0, ..., n-1\}$ dans $\{0, ..., k\}$ (pour $k = n-1$),
3. perm = endofct $\wedge$ inj, par filtrage des injections parmi les endofonctions de $\{0, ..., n-1\}$,
4. perm = endofct $\wedge$ surj, par filtrage des surjections parmi les endofonctions de $\{0, ..., n-1\}$. Cet énumérateur a été détaillé dans la partie 3.3.

Les deux derniers énumérateurs ont été produits. Les permutations ainsi obtenues permettent d'obtenir d'autres familles par filtrage, par exemple les involutions de $\{0, ..., n-1\}$ (invol $\subset$ perm) ou les dérangements (permutations sans point fixe) de $\{0, ..., n-1\}$ (derang $\subset$ perm).

5.1.3. Algorithmes efficaces

La génération dans l'ordre lexicographique est efficace pour les structures suivantes (entre autres) :

— Les fonctions à croissance limitée (rgf) [1, page 235]

— les combinaisons de p éléments parmi n (famille comb) [1, page 178]

Nous avons implanté ces énumérateurs en instanciant le patron de génération dans l'ordre lexicographique présenté dans la partie 4. Les colonnes 4 et 5 montrent que les preuves de ces énumérateurs optimisés sont plus complexes, nécessitant les solveurs CVC3 et CVC4, avec des durées de preuve supérieures à celles des énumérateurs par filtrage. En particulier, l'énumérateur comb se distingue par la difficulté de sa spécification, ce qui implique une durée de preuve plus importante que les autres.

5.2. Validations

Pour valider l'exhaustivité des énumérateurs, nous avons effectué une partie des étapes de validation suivantes, jusqu'à atteindre un niveau de confiance acceptable :

1. Comptage du nombre de structures générées, comparé au nombre attendu, obtenu soit dans une séquence de l'OEIS [15], soit à l'aide de formules de dénombrement connues, implantées sous forme de fonctions C, jusqu'à la limite du plus grand entier positif représentable. Cette limite n'est pas gênante, car on ne peut pas espérer générer autant de structures en un temps raisonnable.
2. Validation que chaque structure obtenue est supérieure à la précédente selon l'ordre lexicographique. Ce n'est qu'à partir de ce niveau qu'on valide l'exhaustivité de l'énumérateur.
3. Validation relative d'un énumérateur par rapport à un autre. Cette validation est souvent facile à mettre en œuvre, d'une part parce que nous obtenons rapidement un énumérateur de référence par filtrage, d'autre part parce que cet énumérateur et l'énumérateur efficace comparé produisent les structures dans le même ordre lexicographique, chaque fois que ce dernier suit le principe de révision du suffixe. Dans ce cas, le stockage des structures énumérées est inutile : il suffit de générer en parallèle les structures avec chaque énumérateur et de tester leur égalité.

Toutes ces validations sont effectuées par taille croissante, jusqu'à une borne raisonnable. La première validation a été effectuée pour toutes les structures de la bibliothèque. La suivante a été implantée pour les énumérateurs efficaces (familles fct, rgf, comb et subset). Enfin, pour la troisième validation, les énumérateurs par filtrage rgf $\subset$ fct et comb $\subset$ fct ont servi d'implantation de référence pour valider les énumérateurs optimisés rgf et comb.

6. Conclusion et perspectives

L'énumération de structures combinatoires par taille croissante jusqu'à une taille donnée peut s'avérer très utile pour tester automatiquement des programmes dont ces structures sont des entrées. Elle permet aussi au combinatoricien de valider expérimentalement des conjectures sur les structures qu'il étudie. Les algorithmes d'énumération efficace posent aussi d'intéressants problèmes de vérification déductive. Pour toutes ces raisons, nous avons entrepris de développer une bibliothèque de programmes d'énumération formellement spécifiés et vérifiés automatiquement, et de mettre au point une méthode pour réduire le coût de leur spécification et de leur vérification déductive.

Cet article présente les premiers résultats, très encourageants. Nous y proposons des patrons généraux pour diverses familles d'énumérateurs de structures, dont l'instanciation produit plus facilement des programmes corrects. En particulier, un patron pour le principe élémentaire de filtrage permet, à partir d'un nombre très restreint de générateurs de familles combinatoires classiques, d'en implanter rapidement beaucoup d'autres. Des programmes plus efficaces sont vérifiés avec davantage de difficultés, mais les greffons de la plateforme Frama-C apportent une aide significative.

S'il existe de nombreux travaux sur le test exhaustif borné [8, 14] et la génération de données structurées pour le test unitaire [3, 9, 5], l'utilisation de la preuve de programmes en combinatoire est beaucoup moins répandue. Dans ce domaine, le travail le plus notable porte sur la fonction de calcul du conjugué d'une partition d'entiers, implantée en C dans le logiciel SCHUR, spécifiée en ACSL et vérifiée avec Frama-C [4].

Il reste beaucoup à faire dans ce domaine. Pour l'instant, on ne spécifie, donc on ne prouve, que la correction des fonctions d'énumération. En complément, nous envisageons de vérifier statiquement que chaque structure produite dans un ordre lexicographique est supérieure à celle à partir de laquelle elle est produite. La propriété d'exhaustivité, selon laquelle toutes les structures d'une taille donnée sont générées, est plus précise, mais semble délicate à spécifier formellement dans un langage comme ACSL. Pour cette propriété, on effectue actuellement des validations, en particulier par comptage des nombres de structures produites.

Enfin, lorsque les limites de la vérification automatique seront atteintes pour ces problèmes, nous pourrons envisager de les traiter en preuve interactive, pour des algorithmes présentant des enjeux suffisants.

Remerciements : les auteurs tiennent à remercier chaleureusement J. Julliand, N. Kosmatov, C. Marché et les relecteurs pour leurs précieux conseils.

Références

[1] J. Arndt. *Matters Computational - Ideas, Algorithms, Source Code [The fxtbook]*. 2010. Published electronically at http ://www.jjj.de.

[2] P. Baudin, P. Cuoq, J. C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*. Published electronically at http ://frama-c.com/acsl.html.

[3] C. Boyapati, S. Khurshid, and D. Marinov. Korat : automated testing based on Java predicates. In *Proceedings of the International Symposium on Software Testing and Analysis, ISSTA 2002, Roma, Italy, July 22-24*, pages 123–133. ACM, 2002.

[4] F. Butelle, F. Hivert, M. Mayero, and F. Toumazet. Formal proof of SCHUR conjugate function. In *Intelligent Computer Mathematics*, volume 6167 of *Lecture Notes in Computer Science*, pages 158–171. Springer, 2010.

[5] M. Gligoric, T. Gvero, V. Jagannath, S. Khurshid, V. Kuncak, and D. Marinov. Test generation through programming in UDITA. In *Proceedings of the 32nd ACM/IEEE International*

Conference on Software Engineering - Volume 1, ICSE 2010, Cape Town, South Africa, 1-8 May, pages 225–234. ACM, 2010.

[6] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. *Formal Aspects of Computing*, To appear.

[7] D. L. Kreher and D. R. Stinson. *Combinatorial algorithms : generation, enumeration, and search.* CRC Press, 1999.

[8] D. Marinov, A. Andoni, D. Daniliuc, S. Khurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical report, MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, 2003.

[9] A. Milicevic, S. Misailovic, D. Marinov, and S. Khurshid. Korat : A tool for generating structurally complex test inputs. In *29th International Conference on Software Engineering (ICSE 2007), Minneapolis, MN, USA, May 20-26*, pages 771–774. IEEE Computer Society, 2007.

[10] G. Petiot, B. Botella, J. Julliand, N. Kosmatov, and J. Signoles. Instrumentation of annotated C programs for test generation. In *14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014, Held as Part of ICSME 2014, Victoria, BC, Canada, September 28-29, 2014. Proceedings. To appear*, pages **–**, 2014.

[11] G. Petiot, N. Kosmatov, A. Giorgetti, and J. Julliand. How test generation helps software specification and deductive verification in Frama-C. In *Tests and Proofs - 8th International Conference, TAP 2014, Held as Part of STAF 2014, York, UK, July 24-25*, pages 204–211, 2014.

[12] F. Ruskey. *Combinatorial Generation Working Version (1j-CSC 425/520)*. 2003.

[13] V. Senni and F. Fioravanti. Generation of test data structures using constraint logic programming. In A. D. Brucker and J. Julliand, editors, *TAP*, volume 7305 of *Lecture Notes in Computer Science*, pages 115–131. Springer, 2012.

[14] K. J. Sullivan, J. Yang, D. Coppit, S. Khurshid, and D. Jackson. Software assurance by bounded exhaustive testing. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2004, Boston, Massachusetts, USA, July 11-14*, pages 133–142. ACM, 2004.

[15] The OEIS Foundation Inc. The On-Line Encyclopedia of Integer Sequences, 2010. Published electronically at http ://oeis.org.