Spécification et vérification formelle d'opérations sur les permutations

Richard Genestier, Alain Giorgetti

Spécification et vérification formelle d'opérations sur les permutations

Richard Genestier et Alain Giorgetti
FEMTO-ST, UMR CNRS 6174 (UBFC/UFC/ENSMM/UTBM)
Université de Franche-Comté, France
firstname.lastname@femto-st.fr

Résumé

Dans le cadre d'un projet plus vaste de formalisation de la combinatoire, nous présentons dans cet article deux opérations sur les permutations, leur implémentation en C sous forme de fonctions modifiant des tableaux d'entiers, et leur spécification formelle en ACSL. Après avoir spécifié des invariants de boucle adaptés, nous prouvons automatiquement que ces fonctions préservent les permutations, avec l'outil Frama-C et son greffon WP.

1 Introduction

Cette étude complète un travail de preuve interactive de la correction d'opérations de construction de cartes combinatoires [DGG16]. Une *carte (combinatoire)* est un triplet (D, R, L) où D est un ensemble fini à $2n$ éléments ($n \geq 0$), R est une permutation sur D et L est une involution sans point fixe sur D, telles que le groupe $\langle R, L\rangle$ engendré par R et L agit transitivement sur D. Dans [DGG16] nous formalisons en Coq [Coq] les notions de permutation et de carte combinatoire, deux opérations sur les permutations et deux opérations de construction de cartes combinatoires, qui utilisent les opérations sur les permutations. Techniquement, nous définissons d'abord ces quatre opérations sur des fonctions quelconques sur D. Puis nous démontrons formellement que ces quatre opérations préservent les permutations et que les deux opérations sur les cartes combinatoires préservent la condition de transitivité. Nous effectuons les démonstrations de manière interactive, à l'aide des tactiques de l'assistant de preuve Coq.

Dans cet article, nous étudions la question de l'automatisation des deux démonstrations portant sur les permutations. Dans ce but, nous représentons une permutation des n premiers entiers naturels par un tableau C de longueur n et nous implémentons les deux opérations sur les permutations par deux fonctions C sur ces tableaux d'entiers. Selon la démarche détaillée dans [GGP15a, GGP15b], nous spécifions en ANSI C Specification Language (ACSL) [BCF+13] que les deux fonctions C peuvent être restreintes aux tableaux qui représentent des permutations : si leurs tableaux en entrée représentent des permutations, alors il en est de même de leur tableau en sortie. On dit aussi que ces fonctions *préservent les permutations*. ACSL est un langage dédié à l'analyse statique de programmes C qui permet de spécifier formellement des contrats qui doivent être vérifiés par le programme. Nous utilisons la plateforme d'analyse de programmes C Frama-C [KKP+15] développée par le CEA LIST et INRIA Saclay, qui utilise le langage de spécification ACSL. Pour la vérification déductive, nous utilisons le greffon WP [BBCD15], qui implémente le calcul de plus faible précondition pour des

programmes C annotés en ACSL, et les solveurs SMT Alt-Ergo [Alt], CVC3 [BT07] et CVC4 [CVC]. Ces solveurs sont utilisés par WP pour déterminer si une formule du premier ordre est satisfaisable.

La partie 2 rappelle quelques notions sur les permutations utiles pour la suite. La partie 3 définit les deux opérations sur les permutations introduites dans [DGG16], puis décrit leur implémentation en C. Leur spécification en ACSL et leur vérification déductive sont détaillées dans la partie 4. La partie 5 conclut cette étude.

2 Permutations

Les permutations sur un ensemble fini forment une famille combinatoire élémentaire mais centrale, largement étudiée dans le domaine de la combinatoire énumérative. Nous proposons une formalisation machine d'opérations basiques sur les permutations, accompagnée de la vérification déductive de leur correction.

Une *permutation* est une bijection sur un ensemble fini, identifié ici à l'ensemble $\{0, \ldots, n-1\}$ des n premiers entiers naturels. La *taille* d'une permutation est la cardinalité n de cet ensemble. L'image de $i \in \{0, \ldots, n-1\}$ par la permutation p sur $\{0, \ldots, n-1\}$ est notée $p(i)$. Appliquer la permutation p_1 puis la permutation p_2 revient à appliquer la permutation $p_1\ p_2$ appelée *composition* ou *produit* de p_1 et p_2. La composition est ici appliquée de gauche à droite, c'est-à-dire que $(p_1\ p_2)(i) = p_2(p_1(i))$. L'ensemble des permutations sur l'ensemble $\{0, \ldots, n-1\}$ muni de l'opération interne de composition est appelé *groupe symétrique* et noté S_n. On peut représenter toute permutation p de S_n sur deux lignes, par la matrice

$$\begin{pmatrix} 0 & 1 & \ldots & i & \ldots & n-1 \\ p(0) & p(1) & \ldots & p(i) & \ldots & p(n-1) \end{pmatrix}$$

ou sur une ligne par le mot $p(0)\ p(1)\ \ldots\ p(n-1)$ sur l'alphabet $\{0, \ldots, n-1\}$. La permutation p est codée en C par le tableau `p` de longueur n tel que `p[i]` $= p(i)$ pour $0 \leq i \leq n-1$, qu'on peut représenter par

`p[0]`	`p[1]`	...	`p[i]`	...	`p[n-1]`

. Ce tableau est appelé *représentation fonctionnelle* de p.

Un *cycle* d'ordre $k \leq n$ est une permutation $p \in S_n$ telle qu'il existe des éléments $i_0, i_1, \ldots, i_{k-1}$ distincts de $\{0, \ldots, n-1\}$ tels que $p(i_0) = i_1, p(i_1) = i_2, \ldots, p(i_{k-2}) = i_{k-1}$, $p(i_{k-1}) = i_0$ et $p(j) = j$ pour tous les autres éléments de $\{0, \ldots, n-1\}$. Un cycle est représenté par sa *notation cyclique* $(i_0\ i_1\ \ldots\ i_{k-1})$. On dit que deux cycles sont *disjoints* s'ils ne modifient pas les mêmes éléments. Toute permutation p de S_n peut être obtenue par produit de cycles disjoints de S_n, appelé *décomposition (cyclique)* de p. Parmi ces décompositions, la *décomposition canonique* de p est le produit de cycles dans lequel la notation de chaque cycle commence par son plus petit élément et les cycles sont écrits dans l'ordre croissant de leur plus petit élément.

Un point fixe x d'une permutation $p \in S_n$ est un élément x de $\{0, \ldots, n-1\}$ tel que $p(x) = x$. Si x est un point fixe de la permutation p, toute décomposition cyclique de p contient le cycle (x), qui ne comporte pas d'autre élément que x.

Exemple 1 *La permutation* $p = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 3 & 5 & 0 & 4 & 2 \end{pmatrix} = 1\ 3\ 5\ 0\ 4\ 2$ *est codée par le tableau C*

1	*3*	*5*	*0*	*4*	*2*

. *Sa décomposition canonique est* $(0\ 1\ 3)\ (2\ 5)\ (4)$. *L'entier* 4 *est son unique point fixe.*

3 Opérations sur les permutations

Dans [DGG16] nous considérons des opérations de construction de cartes combinatoires encodées par des permutations. Pour définir ces opérations, il suffit de savoir insérer un élément dans un cycle d'une permutation et de fusionner deux permutations pour en obtenir une seule. Nous présentons dans les parties 3.1 et 3.2 les opérations d'insertion et de somme directe sur les permutations, ainsi que leurs implémentations par des fonctions C.

En toute rigueur, puisque ces fonctions C agissent sur des tableaux d'entiers C quelconques, cette implémentation généralise à toute fonction définie sur un intervalle d'entiers incluant 0 les deux opérations initialement définies uniquement sur les fonctions bijectives que sont les permutations. Nous ne revenons pas sur ce point dans la suite, et nous ne décrivons cette implémentation que dans le cas où chaque tableau C est la représentation fonctionnelle d'une permutation.

L'opération de composition de deux permutations a également été étudiée mais n'est pas présentée ici, car elle est plus classique et n'est pas utile pour définir les opérations sur les cartes.

3.1 Insertion dans une permutation

Soit p une permutation sur $\{0, \ldots, n-1\}$, vue comme un produit de cycles disjoints, et i un entier naturel entre 0 et n inclus. On appelle *insertion* de i dans p, et on désigne par $insert$, l'opération qui ajoute à p le cycle (n) si $i = n$ et qui insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i \leq n-1$. La permutation qui résulte de cette opération est désignée par $insert(p, i)$.

Exemple 2 *À partir de la permutation* $p = (0\,1\,3)\,(2\,4) = 1\,3\,4\,0\,2 \in S_5$*, on peut définir 6 permutations de* S_6 *par application de l'opération* $insert$ *avec* $i \in \{0, \ldots, 5\}$*. Par exemple,* $insert(p, 0)$ *est la permutation* $(0\,1\,3\,5)\,(2\,4) = 1\,3\,4\,5\,2\,0$*,* $insert(p, 4)$ *est la permutation* $(0\,1\,3)\,(2\,5\,4) = 1\,3\,5\,0\,2\,4$ *et* $insert(p, 5)$ *est la permutation* $(0\,1\,3)\,(2\,4)\,(5) = 1\,3\,4\,0\,2\,5$ *qui ajoute le point fixe* 5 *à* p*.*

Nous présentons dans le listing 1 deux implémentations de cette opération par une fonction C : l'une qui étend la permutation p en place, et l'autre qui duplique cette permutation. Ces deux fonctions utilisent la représentation fonctionnelle de la permutation p.

```
1  void insert_inplace(int p[], int i, int n) {
2   if (0 ≤ i ∧ i ≤ n) {
3    p[n] = i;
4    for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
5   }
6  }
7
8  void insert(int p[], int i, int q[], int n) {
9   if (0 ≤ i ∧ i ≤ n) {
10   q[n] = i;
11   for (int j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
12  }
13 }
```

Listing 1 – Fonctions d'insertion en C.

La première fonction `insert_inplace` prend en paramètres une permutation p, sa taille n et un entier i. Si ces paramètres permettent d'insérer la valeur n dans la permutation p, c'est-à-dire si $0 \leq i \leq n$, le comportement de la fonction est différent selon que i vaut n ou pas. Si $i = n$, la valeur n insérée crée un point fixe dans p (ligne 3). Dans ce cas la boucle de la ligne 4 n'a aucun effet sur p. Si $i < n$, la valeur n est insérée avant la valeur i (ligne 3) et après la valeur $p^{-1}(i)$ dans un cycle de p, par la boucle de la ligne 4.

La seconde fonction `insert` présentée lignes 8-13 effectue la même opération mais en construisant une nouvelle permutation q de taille $n+1$, ce qui nécessite une recopie des valeurs de la permutation p dans q (boucle ligne 11).

3.2 Somme directe de deux permutations

Nous présentons à présent une implémentation en C de l'opération de *somme directe* de deux permutations, bien connue en combinatoire [Kit11]. Cette opération permet de fusionner deux permutations par décalage des valeurs de l'une d'entre elles. Soit p une permutation sur $\{0,\ldots,n-1\}$ et k un entier naturel. On appelle *décalage* de p selon k la permutation p' sur $\{k,\ldots,n+k-1\}$ telle que $p'(i+k) = p(i)+k$. Informellement, l'opération $\oplus$ de somme directe fusionne deux permutations $p_1 \in \{0,\ldots,n_1-1\}$ et $p_2 \in \{0,\ldots,n_2-1\}$ afin d'obtenir une permutation $p_1 \oplus p_2 \in \{0,\ldots,n_1+n_2-1\}$ dont la restriction sur $\{0,\ldots,n_1-1\}$ est égale à p_1 et la restriction sur $\{n_1,\ldots,n_1+n_2-1\}$ est égale au décalage de p_2 selon n_1. Formellement, la somme directe $p_1 \oplus p_2$ de deux permutations $p_1 \in \{0,\ldots,n_1-1\}$ et $p_2 \in \{0,\ldots,n_2-1\}$ est définie par $(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$ et $(p_1 \oplus p_2)(i) = p_2(i-n_1)+n_1$ pour $n_1 \leq i < n_1+n_2$.

Exemple 3 *Considérons les deux permutations* $p_1 = 2\ 1\ 0 = (0\ 2)\ (1) \in S_3$ *et* $p_2 = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4) \in S_5$*, ainsi que la permutation vide notée* $\emptyset$*. Alors* $p_1 \oplus \emptyset = \emptyset \oplus p_1 = p_1$*, le décalage de* p_2 *selon* 3 *est la permutation* $4\ 6\ 7\ 3\ 5 = (3\ 4\ 6)\ (5\ 7)$ *sur* $\{3,\ldots,7\}$*, et* $p_1 \oplus p_2 = 2\ 1\ 0\ 4\ 6\ 7\ 3\ 5 = (0\ 2)\ (1)\ (3\ 4\ 6)\ (5\ 7)$.

Une implémentation de la somme directe sous forme d'une fonction C `sum` est présentée dans le listing 2. La fonction `sum` prend en paramètres une permutation p_1, sa taille n_1, une permutation p_2, sa taille n_2, et construit une nouvelle permutation $p = p_1 \oplus p_2$, résultat de la somme directe de p_1 et p_2. Grâce à une seule boucle, cette fonction recopie les valeurs de p_1 dans p (partie `p1[i]` des affectations de la boucle), effectue le décalage de la permutation p_2 selon n_1, et le recopie dans p à la suite des valeurs de p_1 (partie `p2[i-n1]+n1` des affectations de la boucle).

```
1 void sum(int p1[], int p2[], int p[], int n1, int n2) {
2  for (int i = 0; i < n1+n2; i++) p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
3 }
```

Listing 2 – Somme directe de deux permutations en C.

4 Vérification déductive

Nous pouvons doter les fonctions `insert_inplace`, `insert` et `sum` de contrats et d'annotations, afin de prouver automatiquement qu'elles préservent les permutations.

Il existe différentes manières de caractériser une permutation. Par exemple, bien qu'une permutation soit définie comme une bijection sur un ensemble fini, il est suffisant de la définir comme une endofonction injective ou surjective, puisque sur un ensemble fini l'injectivité et la surjectivité s'impliquent mutuellement. Afin d'effectuer la vérification déductive des fonctions présentées dans la partie 3, nous spécifions une permutation à l'aide du prédicat ACSL `is_perm` présenté dans le listing 3. Ce prédicat caractérise une permutation appartenant à S_n comme une endofonction injective sur $\{0,\ldots,n-1\}$. La propriété caractéristique d'une fonction est exprimée par le prédicat `is_fct` et la propriété d'injectivité par le prédicat `is_linear`. Le prédicat `is_insert` (resp. `is_sum`) paraphrase les instructions de boucle de la fonction `insert` (resp. `sum`).

```
/*@ predicate is_fct(int *a, ℤ b, ℤ c) = ∀ ℤ i; 0 ≤ i < b ⇒ 0 ≤ a[i] < c;
  @ predicate is_linear(int *a, ℤ n) =
  @  ∀ ℤ j; 0 ≤ j < n ⇒ ∀ ℤ k; 0 ≤ k < n ⇒ (j ≠ k ⇒ a[j] ≠ a[k]);
  @ predicate is_perm(int *a, ℤ n) = is_fct(a,n,n) ∧ is_linear(a,n);
  @ predicate is_insert(int *p, int *p1, ℤ b, ℤ c, ℤ d) =
  @  ∀ ℤ i; 0 ≤ i < b ⇒ p[i] == ((p1[i] == c) ? d : p1[i]);
  @ predicate is_sum(int *p, int *p1, int *p2, ℤ b, ℤ c) =
  @  ∀ ℤ i; 0 ≤ i < c ⇒ p[i] == ((i < b) ?  p1[i] : p2[i-b]+b); */
```

Listing 3 – Prédicats ACSL pour les fonctions `insert` et `sum`.

```
1  /*@ requires 0 ≤ i ≤ n;
2    @ requires \valid(p+(0..n-1));
3    @ requires \valid(q+(0..n));
4    @ requires
5    @  \separated(q+(0..n),p+(0..n-1));
6    @ requires is_perm(p,n);
7    @ assigns q[0..n];
8    @ ensures is_perm(q,n+1); */
9
```

```
10 /*@ loop invariant 0 ≤ j ≤ n;
11   @ loop invariant is_insert(q,p,j,i,n);
12   @ loop invariant is_fct(q,j,n+1);
13   @ loop invariant is_linear(q,j);
14   @ loop assigns j, q[0..n-1];
15   @ loop variant n-j; */
```

Listing 4 – Contrat et annotations de boucle de la fonction `insert`.

Le listing 4 présente le contrat (lignes 1-8) et les annotations de boucle (lignes 10-15) de la fonction `insert`. Les annotations de boucle sont à insérer entre les lignes 10 et 11 de la fonction `insert` présentée dans le listing 1. Cette fonction manipulant plusieurs tableaux, il est nécessaire de garantir que ces tableaux sont alloués dans des zones distinctes de la mémoire, à l'aide du prédicat natif `separated` (lignes 4-5). L'invariant ligne 11 spécifie le comportement de la boucle grâce au prédicat `is_insert`. De plus, les invariants de boucle des lignes 12-13 spécifient qu'à chaque itération de boucle, les propriétés caractéristiques d'une permutation sont satisfaites jusqu'à l'indice de boucle.

Le contrat et les annotations de boucle de la fonction `sum` suivent le même principe. Ils sont présentés dans le listing 5. Ces annotations de boucle sont à insérer entre les lignes 1 et 2 de la fonction `sum` présentée dans le listing 2. Les contrats de ces deux fonctions sont prouvés automatiquement, grâce aux invariants de boucle. Ces invariants étant des généralisations "naturelles" des postconditions qu'on veut démontrer, les preuves de ces deux fonctions ne présentent pas de difficulté importante.

```
1  /*@ requires n1 ≥ 0 ∧ n2 ≥ 0;
2    @ requires \valid(p1+(0..n1-1));
3    @ requires \valid(p2+(0..n2-1));
4    @ requires \valid(p+(0..n1+n2-1));
5    @ requires \separated(p1+(0..n1-1),
6    @  p2+(0..n2-1),p+(0..n1+n2-1));
7    @ requires is_perm(p1,n1) ∧
8    @  is_perm(p2,n2);
9    @ assigns p[0..n1+n2-1];
10   @ ensures is_perm(p,n1+n2); */
```

```
11
12 /*@ loop invariant 0 ≤ i ≤ n1+n2;
13   @ loop invariant is_sum(p,p1,p2,n1,i);
14   @ loop invariant is_fct(p,i,n1+n2);
15   @ loop invariant is_linear(p,i);
16   @ loop assigns i, p[0..n1+n2-1];
17   @ loop variant n1+n2-i; */
```

Listing 5 – Contrat et annotations de boucle de la fonction `sum`.

Le listing 6 montre la spécification complète de la fonction `insert_inplace`. Son contrat est plus simple que celui de la fonction `insert`, puisqu'il n'y a pas de tableau `q`. Ses annotations de boucle sont similaires à celles de la fonction `insert`, sauf l'invariant

`loop invariant is_insert(q,p,j,i,n);`

qui est remplacé par deux invariants qui relient les valeurs du tableau dans l'état courant (étiquette `Here`) et dans l'état initial (étiquette `Pre`). Le listing 7 définit les deux prédicats utilisés dans ces invariants. Chaque expression `e` en ACSL peut être écrite `\at(e,L)`, signifiant que l'expression `e` est évaluée dans l'état identifié par l'étiquette `L`. Les étiquettes `Pre` et `Here` sont prédéfinies. Le prédicat `is_insert_inplace` paraphrase le code de la boucle. L'invariant de boucle

`loop invariant is_insert_inplace{Pre,Here}(p,j,i,n);`
l'utilise pour décrire l'évolution de la partie $p[0..j-1]$ du tableau p, entre son indice 0 inclus et l'indice courant j de la boucle exclu. Le prédicat `is_eq_gt` spécifie qu'un suffixe d'un tableau n'est pas modifié entre les états étiquetés par `L1` et `L2`. L'invariant de boucle
`loop invariant is_eq_gt{Pre,Here}(p,j,n);`
l'utilise pour spécifier que, lors de l'exécution de la boucle, la partie $p[j..n-1]$ du tableau p, de l'indice courant de boucle (inclus) à la fin du tableau, n'est pas modifiée.

```
1 /*@ requires 0 ≤ i ≤ n ∧ \valid(p+(0..n-1)) ∧ is_perm(p,n);
2   @ assigns p[0..n];
3   @ ensures is_perm(p,n+1); */
4 void insert_inplace(int p[], int i, int n) {
5  if (0 ≤ i ∧ i ≤ n) {
6   p[n] = i;
7   /*@ loop invariant 0 ≤ j ≤ n;
8     @ loop invariant is_insert_inplace{Pre,Here}(p,j,i,n);
9     @ loop invariant is_eq_gt{Pre,Here}(p,j,n);
10    @ loop invariant is_fct(p,j,n+1);
11    @ loop invariant is_linear(p,j);
12    @ loop assigns j, p[0..n-1];
13    @ loop variant n-j; */
14  for(int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
15 }
16 }
```

Listing 6 – Fonction `insert_inplace` annotée.

```
1 /*@ predicate is_insert_inplace{L1,L2}(int *p, ℤ b, ℤ c, ℤ d) =
2   @  ∀ ℤ k; 0 ≤ k < \at(b,L2) ⇒
3   @    \at(p[k],L2) == ((\at(p[k],L1) == c ) ? d : \at(p[k],L1));
4   @ predicate is_eq_gt{L1,L2}(int *p, ℤ b, ℤ c) =
5   @  ∀ ℤ k; b ≤ k < c ⇒ \at(p[k],L2) == \at(p[k],L1); */
```

Listing 7 – Prédicats ACSL pour la fonction `insert_inplace`.

Grâce à ces invariants, la correction de la fonction `insert_inplace` est prouvée automatiquement par WP. La difficulté ici était de penser à spécifier que la boucle ne modifie pas la partie $p[j..n-1]$ du tableau p.

5 Conclusion

Nous avons implémenté en C deux opérations sur les permutations, spécifié formellement leur comportement et démontré automatiquement que ces implémentations étaient conformes à leur spécification.

Pour les trois opérations (y compris la version `inplace`) le greffon WP de Frama-C Neon-20140301 utilisant les solveurs Alt-Ergo 0.95.2, CVC3 2.4.1 et CVC4 1.3 démontre 54 obligations de preuve en moins d'une minute sur un PC Intel Core i5-3230M à 2,6 GHz sous Linux Ubuntu 14.04. La durée allouée à chaque solveur pour chaque obligation de preuve a été étendue de 10 secondes (valeur par défaut) à une minute.

L'utilisation de la vérification déductive automatique de programmes en combinatoire est pour l'instant peu répandue. Dans ce domaine, le travail le plus proche porte sur la fonction de calcul du conjugué d'une partition d'entiers, implémentée en C dans le logiciel SCHUR, spécifiée en ACSL et vérifiée avec Frama-C [BHMT10]. Un générateur de toutes les solutions du problème des n-reines est spécifié et vérifié avec l'outil Why3 [BFM+13]. La preuve formelle complète nécessite cependant des étapes interactives [Fil12].

Dans [DGG16] nous avons démontré, de manière interactive avec Coq, la correction des opérations sur les cartes qui utilisent les opérations sur les permutations présentées ici. Un défi serait d'étendre la présente étude avec une automatisation de ces démonstrations.

Remerciements. Les auteurs remercient J.-L. Baril et les relecteurs pour leurs suggestions.

Références

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

[BBCD15] P. Baudin, F. Bobot, L. Correnson, and Z. Dargaye. *WP Plug-in Manual*, 2015. `http://frama-c.com/download/frama-c-wp-manual.pdf`.

[BCF+13] P. Baudin, P. Cuoq, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL : ANSI/ISO C Specification Language*, 2013. `http://frama-c.com/acsl.html`.

[BFM+13] F. Bobot, J.-C. Filliâtre, C. Marché, G. Melquiond, and A. Paskevich. The Why3 platform 0.81, 2013. `https://hal.inria.fr/hal-00822856`.

[BHMT10] F. Butelle, F. Hivert, M. Mayero, and F. Toumazet. Formal proof of SCHUR conjugate function. In S. Autexier, J. Calmet, D. Delahaye, P. D. F. Ion, L. Rideau, R. Rioboo, and A. P. Sexton, editors, *Artificial Intelligence and Symbolic Computation (AISC)*, volume 6167 of *LNCS*, pages 158–171. Springer, Heidelberg, 2010.

[BT07] C. Barrett and C. Tinelli. CVC3. In W. Damm and H. Hermanns, editors, *Proceedings of the* 19^{th} *International Conference on Computer Aided Verification (CAV '07)*, volume 4590 of *LNCS*, pages 298–302. Springer, 2007.

[Coq] The Coq Proof Assistant. `http://coq.inria.fr`.

[CVC] CVC4. `http://cvc4.cs.nyu.edu/web/`.

[DGG16] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In B. Aichernig and C. A. Furia, editors, *Tests and Proofs (TAP)*, volume **** of *LNCS*. Springer, Heidelberg, 2016. To appear.

[Fil12] J.-C. Filliâtre. Verifying two lines of C with Why3 : An exercise in program verification. In R. Joshi, P. Müller, and A. Podelski, editors, *Verified Software : Theories, Tools and Experiments (VSTTE)*, volume 7152 of *LNCS*, pages 83–97. Springer, Heidelberg, 2012.

[GGP15a] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In D. Baelde and J. Alglave, editors, *Journées Francophones des Langages Applicatifs (JFLA)*, 2015. `https://hal.inria.fr/hal-01099135`.

[GGP15b] R. Genestier, A. Giorgetti, and G. Petiot. Sequential generation of structured arrays and its deductive verification. In J. C. Blanchette and N. Kosmatov, editors, *Tests and Proofs (TAP)*, volume 9154 of *LNCS*, pages 109–128. Springer, Heidelberg, 2015.

[Kit11] S. Kitaev. *Patterns in permutations and words.* Springer, Berlin, 2011.

[KKP+15] F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-C : a Software Analysis Perspective. *Formal Aspects of Computing*, 27(3) :573–609, 2015.