Combinatoire Formelle avec Why3 et Coq

Alain Giorgetti[1], Catherine Dubois[2] et Rémi Lazarini

[1] Institut FEMTO-ST, Univ. of Bourgogne Franche-Comté, CNRS, Besançon, France
alain.giorgetti@femto-st.fr
[2] Samovar, ENSIIE, CNRS, Évry, France
catherine.dubois@ensiie.fr

Résumé

Nous présentons une approche formelle de la combinatoire, qui applique les méthodes du test unitaire et de la preuve formelle à des problèmes et algorithmes de combinatoire énumérative ou bijective. Cette combinatoire formelle utilise la plateforme de vérification déductive Why3, pour soumettre les démonstrations à des prouveurs automatiques ou les traduire dans le langage des assistants de preuve. Elle utilise aussi l'assistant de preuve Coq, dont le langage permet de formaliser des démonstrations et de les mener de manière interactive.

1 Introduction

La combinatoire est la branche des mathématiques qui étudie diverses configurations d'un ensemble fini d'objets, comme les permutations ou les mots. La combinatoire énumérative liste et dénombre ces structures combinatoires selon leur *taille*, qui est (le plus souvent) le nombre d'objets qui les composent. La combinatoire bijective établit des bijections structurelles non triviales entre diverses familles de structures combinatoires.

Nous désignons par *combinatoire formelle* l'application des méthodes formelles du génie logiciel à la recherche en combinatoire, en particulier pour prouver formellement des théorèmes de combinatoire énumérative ou bijective, ou des propriétés de programmes de comptage, d'énumération ou de transformation bijective de structures combinatoires.

D'importants travaux de formalisation des mathématiques ont produit des démonstrations formelles de théorèmes célèbres, comme le théorème des quatre couleurs en théorie des graphes [25] ou le théorème de Feit-Thompson en théorie des groupes [26]. Ces tours de force mobilisent de nombreux chercheurs pendant plusieurs années et portent sur des résultats majeurs, préalablement établis de manière informelle. De manière complémentaire, nous souhaitons encourager la pratique de la formalisation lors de recherches mathématiques plus modestes, dans des délais plus courts et sur des problèmes ouverts, afin que les efforts investis en formalisation et recherche de preuve produisent des gains de productivité de nouveaux théorèmes, et de confiance dans leur correction ainsi vérifiée. Des interactions et collaborations récentes avec des combinatoriciens nous ont convaincus que cette approche formelle pouvait bien convenir à certains sujets de recherche actuels sur les permutations ou les cartes combinatoires [2, 15, 24].

Nous étudions dans cet article l'adéquation des environnements Why3 [5] et Coq [10] à cette pratique de la combinatoire formelle. Why3 [5] est une plateforme de vérification déductive de programmes écrits en WhyML, un dialecte de ML avec certains traits fonctionnels, comme les types algébriques polymorphes ou le filtrage, mais aussi des traits impératifs, comme les boucles ou les enregistrements avec des champs mutables. Why3 traduit un programme et sa spécification en un ensemble de conditions logiques de vérification, qui peuvent être soumises à des prouveurs automatiques, tels que les solveurs modulo théorie (SMT), comme Alt-Ergo [1], CVC4 [12] ou Z3 [13]. Si ces prouveurs échouent, les conditions de vérification peuvent être prouvées interactivement au sein d'un assistant de preuve comme Coq.

Dans un assistant de preuve, il est précieux de se convaincre qu'un lemme est correct, avant d'investir du temps dans sa démonstration interactive. Dans ce but, nous avons complété Coq avec un outil de test automatique de conjectures, qui utilise des programmes d'énumération de données de test écrits en OCaml (partie 3). Afin d'accroître la confiance dans ces programmes, nous proposons de les produire par extraction de programmes WhyML dont la correction et la complétude sont démontrées formellement avec Why3 et Coq (partie 2). Lorsque ces programmes énumèrent des structures combinatoires, cette preuve formelle est une activité de combinatoire formelle, qui complète, voire remplace, une preuve informelle d'un ouvrage de combinatoire énumérative. Notre troisième contribution est une étude de cas originale, composée des deux premières étapes d'une démonstration formelle d'une bijection entre une structure combinatoire particulière – les nombres factoriels – et les permutations de même taille (partie 4). La première étape est la certification d'un programme d'énumération de tableaux factoriels, utile pour tester toute conjecture sur les nombres factoriels. La deuxième étape est une preuve formelle de correction de la traduction des tableaux factoriels en permutations.

Le code présenté est disponible dans la version 2.2 de notre outil CUT[1] (*Coq Unit Testing*) de test de conjectures Coq.

2 Preuve de programmes d'énumération

Cette partie présente une méthode de spécification, programmation et preuve formelle de programmes d'énumération avec Why3. Elle généralise une étude précédente limitée aux permutations [23] et adapte à Why3 une méthode de preuve de programmes d'énumération écrits en C et spécifiés en ACSL [22].

Nous étudions la famille des programmes d'énumération de structures combinatoires appelés *générateurs lexicographiques*, car ils énumèrent toutes les structures de même taille dans l'ordre lexicographique (spécifié en WhyML dans la partie 2.1). Par souci de simplicité, notre exposé est limité aux structures combinatoires représentées par un tableau à valeurs dans un intervalle borné d'entiers. La partie 2.2 introduit l'exemple fil rouge des tableaux à valeurs dans un intervalle de la forme $[0..b-1]$. Les parties 2.3 et 2.4 décrivent respectivement une formalisation des générateurs lexicographiques et de leurs propriétés en WhyML. La partie 2.5 décrit une bibliothèque de générateurs certifiés.

2.1 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 $a[i] < b[i]$.

Le listing 1 reproduit un extrait du module `Lex` que nous proposons pour spécifier formellement l'ordre lexicographique en WhyML. Par souci de simplicité, l'ensemble A est ici l'ensemble des entiers relatifs du type `int` de Why3, et l'ordre $<$ sur A est le prédicat

```
val predicate (<) int int : bool
```

défini en WhyML de telle sorte que sa fermeture réflexive

```
let predicate (≤) (x y : int) = x < y || x = y
```

[1] http://members.femto-st.fr/alain-giorgetti/en/coq-unit-testing.

```
module Lex
  use import int.Int
  use import array.Array
  use import array.ArrayEq

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

  predicate le_lex_sub (a1 a2: array int) (l u: int) = lt_lex_sub a1 a2 l u ∨
    array_eq_sub a1 a2 l u
  predicate le_lex (a1 a2: array int) = a1.length = a2.length ∧ le_lex_sub a1 a2 0 a1.length
end
```

Listing 1: Spécification de l'ordre lexicographique.

soit un ordre total. Le prédicat `array_eq_sub` est défini dans la bibliothèque standard de Why3 par

```
  predicate array_eq_sub (a1 a2: array 'a) (l u: int) = a1.length = a2.length ∧
    0 ≤ l ≤ a1.length ∧ 0 ≤ u ≤ a1.length ∧ map_eq_sub a1.elts a2.elts l u
```

avec

```
  predicate map_eq_sub (a1 a2 : map int 'a) (l u : int) =
    ∀ i:int. l ≤ i < u → a1[i] = a2[i]
```

L'expression `(array_eq_sub a1 a2 l u)` formalise que les tableaux a_1 et a_2 sont de même longueur et que les sous-tableaux $a_1[l..u-1]$ et $a_2[l..u-1]$ sont égaux. Le prédicat `lt_lex` formalise sur les tableaux d'entiers Why3 l'ordre lexicographique strict $\prec$ induit par l'ordre strict `<` sur les entiers Why3. Ce prédicat est généralisé à tout sous-tableau par le prédicat `lt_lex_sub`. Les prédicats `le_lex` et `le_lex_sub` sont leur fermeture réflexive respective.

Totalité. L'ordre lexicographique $\prec$ est total si l'ordre $<$ sur A est total, ce qui est le cas pour l'ordre `<` sur `int` de Why3. Nous démontrons formellement cette propriété de totalité, importante car elle permet de démontrer la complétude de certains générateurs lexicographiques (la propriété de complétude d'un générateur est définie plus loin, dans la partie 2.4).

La *totalité* de l'ordre lexicographique $\prec$ est la propriété $(\forall a\ b.\ a \not\prec b \Rightarrow b \preceq a)$ selon laquelle, si le tableau a n'est pas inférieur au tableau b, alors il lui est supérieur ou égal. Nous la généralisons à tout sous-tableau entre les indices l et $u-1$ inclus, par le lemme

```
  lemma total_order: ∀ a b: array int, l u: int.
    0 ≤ l < u ≤ a.length = b.length ∧ not (lt_lex_sub b a l u) → le_lex_sub a b l u
```

en WhyML. Ce lemme est démontré automatiquement, après l'ajout de la *lemma function*

```
  let rec lemma not_array_eq_sub (a b: array int) (l u: int)
    requires { 0 ≤ l < u ≤ a.length = b.length }
    requires { not (array_eq_sub a b l u) }
    variant  { u - l }
    ensures  { ∃ i:int. l ≤ i < u ∧ array_eq_sub a b l i ∧ a[i] ≠ b[i] }
  = if a[l] = b[l] then not_array_eq_sub a b (l+1) u
```

Une *lemma function* est un lemme dont l'énoncé est un contrat de fonction et la preuve est un programme *pur*, qui termine toujours et sans effets de bord. Ce programme démontre le lemme selon lequel la précondition de la fonction implique sa postcondition. Cette fonctionnalité de Why3 est présentée et utilisée, par exemple, dans la partie 3.1 de [9]. Nous avons ici une preuve

du lemme

```
lemma not_array_eq_sub: ∀ a b: array int, l u: int.
  0 ≤ l < u ≤ a.length = b.length ∧ not (array_eq_sub a b l u) →
  ∃ i:int. l ≤ i < u ∧ array_eq_sub a b l i ∧ a[i] ≠ b[i]
```

qui stipule que si les tableaux $a[l..u-1]$ et $b[l..u-1]$ sont différents alors il existe un indice i tel que les sous-tableaux $a[l..i-1]$ et $b[l..i-1]$ sont égaux et $a[i] \neq b[i]$. La fonction récursive `not_array_eq_sub` démontre ce lemme par récurrence sur la longueur des sous-tableaux considérés. Ce lemme et sa démonstration pallient le manque de support du raisonnement par induction dans les solveurs SMT.

2.2 Exemple des tableaux bornés

Comme exemple fil rouge de structure combinatoire à énumérer, considérons la famille des tableaux à valeurs dans l'intervalle initial d'entiers naturels $[0..b-1]$, pour une certaine *borne* $b > 0$. Cette famille des tableaux *bornés* (ou *initiaux*) – baptisée `barray` en anglais, pour "*bounded array*" – est une structure combinatoire, car il y a un nombre fini de tableaux initiaux de taille n et de borne b. Son prédicat caractéristique peut être formalisé par le prédicat

```
predicate is_barray (a:array int) (b:int) =
  ∀ i:int. 0 ≤ i < a.length → 0 ≤ a[i] < b
```

2.3 Fonctions d'énumération

Les programmes d'énumération modifient un état, appelé *curseur*, dont le type est défini par

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

en WhyML. Le champ `current` stocke la dernière structure énumérée. Ici c'est un tableau d'entiers mutable, mais d'autres types peuvent être utilisés. Le champ booléen `new` prend la valeur `false` si et seulement si la structure stockée dans le champ `current` a déjà été énumérée. Ce curseur est une variante du curseur proposé par Filliâtre et Pereira pour l'itération [19].

Considérons une famille `X` de structures combinatoires, caractérisée par un prédicat

```
predicate is_X (a: array int) (...) = ...
```

où `(...)` représente d'éventuels paramètres supplémentaires. Un *générateur lexicographique* pour cette famille `X` est composé de deux fonctions : une fonction d'initialisation et une fonction de passage au suivant.

La fonction d'*initialisation*

```
create_cursor (n: int) : cursor
```

retourne un curseur initialisé avec le plus petit tableau de taille n dans cette famille `X`, selon l'ordre lexicographique $\prec$.

La fonction de *passage au suivant*

```
next (c : cursor) : unit
```

remplace la structure a stockée dans le curseur c par la structure de la famille `X` qui la suit immédiatement selon l'ordre lexicographique, si la structure a n'est pas maximale. Sinon, la fonction donne la valeur `false` au champ booléen `new` du curseur c.

Ce couple de fonctions est appelé générateur *à petits pas*, pour le distinguer du *générateur à grands pas* du listing 2, qui réalise l'énumération exhaustive à l'aide de ces deux fonctions. Ce programme produit une à une toutes les structures combinatoires d'une taille donnée n (dans

```
let big_step_gen (f: list int → 'a) (n: int) : unit =
  requires { n ≥ 0 }
  diverges
  let c = create_cursor n in
  while c.new do
    let a = c.current in
    f (to_list a 0 a.length);
    next c
  done
```

Listing 2: Générateur à grands pas.

le curseur c) et leur applique le même traitement (une fonction f donnée). Techniquement, ces tableaux d'entiers sont d'abord convertis en listes par la fonction `to_list`, car Why3 ne permet pas l'application directe d'une fonction sur un tableau d'entiers.

Ce générateur à grands pas est un cas particulier d'itérateur d'ordre supérieur, écrit ici dans le style impératif. Les qualificatifs "à grands pas" et "à petits pas" ont été introduits par Filliâtre et Pereira [18] pour distinguer les itérateurs qui contrôlent l'itération de ceux qui laissent le contrôle à l'utilisateur. Nos générateurs sont des cas particuliers de ces deux sortes d'itérateurs. Ils parcourent des collections qui ne sont pas stockées en mémoire mais dont chaque donnée est produite en place et utilisée à la volée.

```
let create_cursor (n b: int) : cursor
  requires { n ≥ 0 }
  requires { b > 0 }
= let a = make n 0 in
  { current = a; new = true; bound = b }

let next (c: cursor) : unit
= let a = c.current in
  let n = a.length in
  let b = c.bound in
  let r = ref (n-1) in
  while !r ≥ 0 && a[!r] = b-1 do
    r := !r - 1
  done;
  if (!r < 0) then
    c.new ← false
  else begin
    a[!r] ← a[!r] + 1;
    for i = !r+1 to n-1 do
      a[i] ← 0
    done;
    c.new ← true
  end
```

Listing 3: Énumération des tableaux bornés en WhyML.

Exemple. Le listing 3 présente des fonctions d'énumération pour les tableaux bornés. Pour cette structure, le curseur est complété avec un champ

```
bound: int;
```

qui stocke la borne des tableaux. La fonction `create_cursor` construit le plus petit tableau borné a de longueur $n \geq 0$, tel que $a[i] = 0$ pour tout indice i, sous la (pré)condition que la borne b soit strictement positive, car sinon aucun tableau de cette nature n'existe. La première boucle de la fonction `next` détermine l'indice $!r$ le plus élevé tel que $a[!r] < b - 1$ puisse être incrémenté. Si $!r$ atteint -1, le tableau a est maximal et l'énumération est terminée. Sinon, $a[!r]$ est incrémenté et la deuxième boucle remplace par 0 la valeur de chaque élément du sous-tableau $a[!r + 1..n - 1]$.

2.4 Preuve de propriétés des générateurs lexicographiques

Nous souhaitons spécifier et démontrer formellement que chaque générateur lexicographique satisfait les propriétés suivantes de correction et de complétude.

Correction. La *correction* est la propriété que chaque structure générée de la famille `X` satisfait son prédicat caractéristique `is_X`. Nous introduisons le prédicat

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

et nous spécifions la correction par les postconditions respectives

```
ensures { sound result }
```

et

```
ensures { sound c }
```

des fonctions `create_cursor` et `next`, en ajoutant la précondition

```
requires { sound c }
```

à la fonction `next`. Ceci exige que la fonction d'initialisation construise une structure de la famille `X` et que la fonction de passage au suivant transforme toute structure de la famille `X` en une structure de la même famille.

Complétude. La propriété de *complétude* exige que le générateur à grands pas produise toutes les structures d'une taille donnée. Puisque les structures sont générées selon l'ordre lexicographique total $\prec$, la complétude du générateur à grands pas résulte de la conjonction des trois propriétés suivantes :

1. la fonction d'initialisation génère la plus petite structure selon $\prec$ (propriété *min_lex*),
2. la fonction de passage au suivant transforme chaque structure en la structure immédiatement supérieure selon $\prec$ (propriété d'*incrémentation*), et
3. le générateur à grands pas termine lorsque le curseur `c` contient la plus grande structure selon $\prec$ (propriété *max_lex*).

Nous détaillons la formalisation de ces trois propriétés :

1. La propriété *min_lex*, qui impose que la première structure générée soit la plus petite structure selon l'ordre lexicographique, est spécifiée par la postcondition

   ```
   ensures { min_lex result.current }
   ```

   dans le contrat de la fonction `create_cursor`, avec le prédicat

   ```
   predicate min_lex (a: array int) = ∀ b: array int.
     a.length = b.length ∧ is_X b → le_lex_sub a b 0 a.length
   ```

2. La propriété d'*incrémentation* spécifie qu'une structure a_2 générée par la fonction `next` à partir d'une 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 telle que $a_1 < a_3 < a_2$. Cette propriété est spécifiée par la postcondition

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

   de la fonction `next`, avec

   ```
   predicate inc (a1 a2: array int) = lt_lex a1 a2 ∧ ∀ a3: array int.
     lt_lex a1 a3 ∧ is_X a3 → le_lex a2 a3
   ```

3. La propriété *max_lex*, qui exige que la dernière structure générée soit la plus grande selon l'ordre lexicographique, est spécifiée par la postcondition

```
ensures { not c.new → max_lex result.current }
```

dans le contrat de la fonction `next`, avec le prédicat

```
predicate max_lex (a: array int) = ∀ b: array int.
  a.length = b.length ∧ is_X b → le_lex_sub b a 0 a.length
```

2.5 Catalogue de générateurs certifiés

Genestier, Petiot et Giorgetti [21, 22] ont développé en C et spécifié en ACSL une bibliothèque de programmes d'énumération de structures combinatoires, nommée enum. La version la plus récente de la bibliothèque enum contient 24 générateurs et un mécanisme de construction de générateurs par filtrage des données produites par un générateur plus général.

Les structures de données générées sont les tableaux bornés, les parties d'un ensemble à n éléments, les tableaux bornés triés, les combinaisons de p éléments parmi n, les injections de $[0..n-1]$ dans $[0..k]$ (pour $n \leq k+1$), les surjections de $[0..n-1]$ dans $[0..k]$ (pour $n \geq k+1$), les permutations, les involutions, les dérangements (permutations sans point fixe), les fonctions à croissance limitée, les involutions sans point fixe, les permutations connectées, les involutions connectées quelconques et les involutions connectées sans point fixe. Toutes ces structures sont définies dans la thèse de Genestier [20]. Pour certaines d'entre elles, plusieurs générateurs sont proposés et vérifiés.

Les propriétés vérifiées avec la plateforme Frama-C et son greffon de vérification déductive WP sont la correction et le *progrès*, qui est la propriété que chaque structure générée par la fonction de passage au suivant est supérieure à la précédente, selon l'ordre lexicographique. Cependant, la propriété de complétude n'a pas pu être démontrée formellement dans cet environnement. Cette propriété plus complexe inclut une quantification sur les tableaux. Pour faciliter sa démonstration, Giorgetti et Lazarini ont adapté l'un des programmes – énumérant des permutations – au langage WhyML et à sa structure de tableaux mutables [23]. Ces travaux antérieurs sont respectivement disponibles dans les versions 1.0[2] et 1.1[3] de la bibliothèque enum.

Depuis, deux autres générateurs ont été certifiés avec Why3 : le générateur de tableaux bornés présenté dans la partie 2.2 et le générateur de tableaux factoriels présenté dans la partie 4.2. Ce catalogue de générateurs certifiés avec Why3 est disponible en *open source*, pour être étendu et dépasser la portée de la version 1.0 de la bibliothèque enum en C/ACSL. Les propriétés des générateurs de tableaux bornés, factoriels et de permutations ont été prouvées avec les solveurs Alt-Ergo 1.30, CVC4 1.5 et Z3 4.7.1 – parfois en augmentant la limite de temps par preuve de 5 à 10 secondes – sauf la complétude du générateur de permutations, qui a nécessité deux preuves interactives, réalisées avec Coq 8.7.0, pour un total de 177 lignes de preuve.

3 Test de conjectures

Cette partie présente une application majeure des programmes d'énumération de structures combinatoires, qui justifie leur développement et leur certification avec Why3. Cette application est le test des propriétés qui ne sont pas démontrées automatiquement dans un délai

[2] Archive `enum-1.0.tar.gz` sur la page http://members.femto-st.fr/alain-giorgetti/en.
[3] https://github.com/alaingiorgetti/enum.

raisonnable. Les démontrer interactivement est une activité délicate, qui requiert une connaissance approfondie des tactiques de preuve et de la bibliothèque de définitions et théorèmes de l'assistant de preuve utilisé. Tester une propriété avant d'en commencer une démonstration interactive permet de se convaincre qu'elle est correcte, ou sinon de gagner un temps précieux en détectant rapidement une faille de raisonnement ou une erreur de formulation.

Le *test de propriété* (PBT, pour *Property-Based Testing*) est la recherche d'un contre-exemple pour une propriété d'un programme en cours de vérification. Il est populaire pour les langages fonctionnels, notamment avec l'outil QuickCheck [8] dans Haskell. Le PBT a également été adapté à des assistants de preuve, comme Isabelle [4], Agda [17], PVS [32], FoCaLiZe [7] et plus récemment Coq [33], avec l'outil de test aléatoire QuickChick. Dans ce cadre des assistants de preuve, nous parlerons de *test de conjecture*.

Parmi les méthodes de test automatique, le test exhaustif borné (BET, pour *Bounded Exhaustive Testing*) d'une fonction ou propriété consiste à générer toutes ses données d'entrée jusqu'à une certaine taille. Le BET est particulièrement bien adapté aux structures combinatoires, car ses contre-exemples sont toujours de taille minimale (ce qui facilite le déboguage), sa couverture est bien identifiée (puisqu'il constitue une preuve par énumération de tous les cas jusqu'à la borne de test), et une donnée de petite taille suffit souvent pour révéler une erreur [27]. Ainsi, le BET est complémentaire du test aléatoire, plus adapté aux domaines de données de plus grande taille.

Certains outils de test généraux sont capables de générer des données ou de dériver des générateurs à partir de la définition de ces données, selon diverses techniques, détaillées dans les introductions de travaux récents sur ce sujet [29, 11]. Par exemple, Dubois, Giorgetti et Genestier ont complété QuickChick avec une première approche de test exhaustif borné fondée sur des générateurs dérivés de définitions des données en programmation logique (Prolog) [15].

Cependant, pour certaines structures de données, ces outils peuvent être trop lents, échouer dans la dérivation d'un générateur ou dériver des générateurs peu efficaces. Il devient alors pertinent de concevoir un *générateur dédié* à chaque structure, voire de le certifier pour l'intégrer avec confiance dans un outil de test. Par exemple, Bowles et Caminati ont vérifié un algorithme d'énumération de structures d'événements [6]. Dubois et Giorgetti ont développé l'outil CUT, qui ajoute à Coq les commandes SmallCheck et SmallCheckWhy3 de test exhaustif borné avec des programmes d'énumération dédiés, respectivement définis dans les langages de Coq et de Why3 [14].

Cette partie décrit l'intégration dans la commande SmallCheckWhy3 d'un programme d'énumération en WhyML, éventuellement certifié avec Why3, comme présenté dans la partie 2. Nous détaillons ainsi une présentation précédente [14] en l'illustrant avec un exemple original.

Considérons une conjecture Coq de la forme

$$\forall \mathsf{x} : \mathsf{T},\ \mathsf{precondition}\ \mathsf{x} \rightarrow \mathsf{conclusion}\ \mathsf{x}, \tag{1}$$

où precondition et conclusion sont des prédicats logiques de type T → Prop. Pour tester cette conjecture, nous devons disposer d'une fonction booléenne – nommée conclusionb – exécutable et sémantiquement équivalente au prédicat logique conclusion.

L'exécution de la commande Coq

SmallCheckWhy3 *path* (it_X *size*) conclusionb.

réalise le BET de cette conjecture à l'aide d'un générateur OCaml de termes de type T satisfaisant la propriété precondition, si *path* est un chemin relatif vers le dossier du code source OCaml du générateur, si it_X est un itérateur Coq qui interface ce générateur OCaml avec Coq, et si *size* est la taille des données à générer. Nous nous intéressons ici au cas où le générateur OCaml est obtenu par extraction d'un générateur écrit en WhyML et certifié avec Why3.

Exemple. Illustrons cette commande Coq avec l'exemple de la structure de liste d'entiers naturels bornés (version Coq des tableaux bornés WhyML de la partie 2.2). Dans ce cas, T dans la conjecture (1) est le type list nat des listes d'entiers naturels en Coq. La famille des listes Coq dont les éléments sont des entiers naturels strictement inférieurs à une borne donnée b est caractérisée par le prédicat logique is_blist, qui est défini inductivement par

```
Inductive is_blist (b : nat) : list nat → Prop :=
| Blist_nil : is_blist b nil
| Blist_cons : ∀ v l, v < b → is_blist b l → is_blist b (v :: l).
```

et qui constitue la precondition de la conjecture (1). Ce prédicat n'est pas exécutable, mais nous pouvons prouver son équivalence avec la fonction booléenne is_blistb suivante :

```
Fixpoint is_blistb (b : nat) (l : list nat) :=
  match l with
    nil    ⇒ true
  | v :: l' ⇒ (ltb v b) && (is_blistb b l')
  end.
Lemma is_blist_dec : ∀ b l, (is_blistb b l = true ↔ is_blist b l).
```

Nous voulons tester la conjecture Coq

```
∀ (l : list nat) (b : nat), is_blist b l → is_blist (rev b l).
```

qui affirme que le miroir de toute liste l bornée par b est une liste bornée par la même valeur b.

Supposons que le générateur Why3 de tableaux bornés de la partie 2.3 soit défini dans le module Enum d'un fichier nommé Barray.mlw. Son extraction en OCaml par Why3 est un fichier Barray__Enum.ml, accompagné de fichiers auxiliaires. Soit `../barray` le dossier d'extraction.

```
Parameter cursor : Type.
Parameter create : nat → nat → cursor.
Parameter next_list : cursor → (option (list nat)) * cursor.
Parameter hasnext_list : cursor → bool.

Definition it_blist (n b : nat) := {|
 st_t := cursor;
 start := create n b;
 next := next_list;
 hasnext := hasnext_list |}.

Extract Constant cursor ⇒ "Barray__Enum.cursor".
Extract Constant create ⇒ "fun n → fun b →
  Barray__Enum.create_cursor (Why3extract.Why3__BigInt.of_int n)
  (Why3extract.Why3__BigInt.of_int b)".
Extract Constant next_list ⇒ "(fun c →
  Barray__Enum.next c;
  let a = c.current in (Some (to_list a), c))".
Extract Constant hasnext_list ⇒ "Barray__Enum.has_next".
```

Listing 4: Itérateur Coq sur les listes bornées.

Le listing 4 reproduit le code Coq d'un itérateur it_blist sur les listes bornées, interfacé avec le générateur OCaml de tableaux bornés. Les commandes d'extraction peuvent utiliser

des fonctions Why3extract.* issues de l'extraction Why3. Ici, elles utilisent aussi une fonction `to_list` de conversion de tableaux en listes. De plus, une fonction booléenne

```
let has_next (c: cursor) : bool = c.new
```

qui détermine si un curseur a un successeur doit être ajoutée au générateur WhyML.

Alors, la commande Coq

```
SmallCheckWhy3 "../barray" ( it_blist  7 4) (fun l ⇒ is_blistb  4 (rev  l )).
```

teste la conjecture avec toutes les listes de longueur 7 dont les éléments sont des entiers dans l'intervalle [0..3]. Techniquement, la commande SmallCheckWhy3 est ajoutée à l'outil de test aléatoire QuickChick, afin de ré-utiliser son mécanisme d'extraction OCaml pour exécuter les cas de test.

4 Étude de cas

Cette partie illustre la démarche de combinatoire formelle par une étude de cas originale, qui utilise Why3 pour démontrer un théorème lié aux permutations.

En 1888, C.-A. Laisant [28] introduit un système de numération, appelé *numération factorielle*, qui distingue $n!$ nombres à n chiffres, pour représenter les $n!$ permutations de n éléments. Un *code de permutation* est une transformation bijective de ces nombres en permutations de taille n. Divers codes de permutation sont connus et largement étudiés en combinatoire [30, 16, 31, 34, 3]. Nous souhaitons étudier formellement ces codes, c'est-à-dire les programmer et démontrer formellement certaines de leurs propriétés. La présente étude initie ce programme de recherche, en proposant un premier code de permutation programmé en WhyML, et une démonstration formelle qu'il ne produit que des permutations.

Les nombres de Laisant sont parfois dits *subexcédants*, avec les deux orthographes *subexceedant* [16] et *subexcedant* [34] en anglais. Nous les appelons plus simplement *(nombres) factoriels* en français, et *factorial numbers* ou *factorials* en anglais.

Les étapes de cette étude sont : une spécification formelle de la propriété des nombres factoriels (partie 4.1), une implémentation d'un générateur lexicographique de factoriels en WhyML, une spécification et une preuve formelle de ses propriétés de correction et de complétude (partie 4.2), une implémentation du code de permutation en WhyML, en tant que transformation de tableaux de même taille (partie 4.3), et enfin une spécification et une preuve formelle de la propriété que ce code transforme tout factoriel en permutation (partie 4.4).

4.1 Factoriels

Un mot $f_{n-1} \dots f_0$ de longueur n est un *(nombre) factoriel* si et seulement si sa lettre f_i est un entier naturel inférieur ou égal à i, pour $0 \leq i \leq n-1$. Par exemple, 23110 est un factoriel, tandis que 3020 n'est pas un factoriel, car son deuxième chiffre le plus à droite est 2, qui n'est pas inférieur ou égal à 1. En particulier, le dernier chiffre f_0 d'un factoriel est toujours zéro. Le nombre factoriel $f_{n-1} \dots f_0$ est la *numération factorielle* de l'entier naturel $\Sigma_{i=0}^{n-1} f_i\ i!$ et cette numération est une bijection entre nombres factoriels de longueur n et nombres entiers dans l'intervalle $[0..(n!-1)]$.

Nous représentons ces mots par des tableaux mutables WhyML et nous spécifions leur prédicat caractéristique de *(tableau) factoriel* avec les définitions suivantes :

```
predicate is_fact_sub (a:array int) (l u:int) = ∀ i:int.
  l ≤ i < u → 0 ≤ a[i] ≤ i
```

qui spécifie que le sous-tableau $a[l..u-1]$ est factoriel, et

```
predicate is_fact (a:array int) = is_fact_sub a 0 a.length
```

qui spécifie que le tableau a en paramètre représente un nombre factoriel.

Le chiffre f_i du nombre factoriel $f_{n-1} \dots f_0$ est stocké dans la case $a[i]$ du tableau factoriel correspondant. Remarquons cependant que, selon les usages d'écriture des nombres et des tableaux, le nombre factoriel $f_{n-1} \dots f_0$ s'écrit avec le chiffre $f_0 = 0$ de poids faible (égal à 0!) à droite, tandis que son tableau factoriel

0	1	2	...	$n-2$	$n-1$
f_0	f_1	f_2	...	f_{n-2}	f_{n-1}

est présenté dans le sens inverse, avec f_0 comme chiffre le plus à gauche. Pour éviter cette confusion, nous ne traitons plus de nombres factoriels, mais uniquement de tableaux factoriels, en présentant leur contenu $f_0 \dots f_{n-1}$ dans l'ordre croissant des indices.

4.2 Générateur lexicographique de tableaux factoriels

Un générateur lexicographique de tableaux factoriels est reproduit dans le listing 5. La fonction `create_cursor` construit le plus petit tableau factoriel a, tel que $a[i] = 0$ pour tout indice i. Le contrat de la fonction `next` spécifie ses propriétés de *correction*, d'*incrémentation* et de dernier tableau généré maximal. Les variables auxiliaires a et n désignent respectivement la structure courante et sa taille. Le label `'L` permet de décrire par `(at a 'L)` le tableau `a` tel qu'il est au niveau du label. La boucle `while` implémente la recherche de l'indice $!r$ de révision du tableau. Cette boucle est spécifiée par deux invariants, qui bornent les valeurs de $!r$ et assurent que le sous-tableau $a[r+1..n-1]$ est maximal, avec le prédicat `is_id_sub` défini par

```
predicate is_id_sub (a:array int) (l u:int) = ∀ i:int. l ≤ i < u → a[i] = i
```

Un variant permet de démontrer la terminaison de cette boucle. Si l'indice de révision $!r$ vaut -1, le tableau est maximal et la génération est terminée. Sinon, on a $a[!r] < !r$, ce qui permet d'incrémenter $a[!r]$, puis de remplir le sous-tableau $a[r+1..n-1]$ avec des 0, grâce à la boucle `for`. Ses trois premiers invariants bornent son indice i et assurent que la valeur de $a[!r]$ n'est pas modifiée par la boucle et que le sous-tableau $a[!r+1..i-1]$ ne contient que des 0, avec le prédicat `cte_sub` défini par

```
predicate cte_sub (a:array int) (l u:int) (b:int) = ∀ i:int. l ≤ i < u → a[i] = b
```

Avec la définition

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

le dernier invariant assure que le tableau courant est strictement supérieur au tableau en entrée, tout en précisant que le plus petit indice auquel les valeurs des deux tableaux diffèrent est $!r$. Cette précision permet aux solveurs SMT de choisir $!r$ pour démontrer la propriété d'incrémentation.

4.3 Code de permutation en WhyML

Soit n un entier naturel et f un tableau factoriel de longueur n. L'algorithme de la figure (1a) transforme le factoriel f en une permutation p, également représentée dans un tableau de longueur n. Dans cet algorithme, $t(x)$ désigne le $(x+1)$-ème élément de la liste ou du tableau t. La figure (1b) illustre l'algorithme avec les valeurs successives des variables, pour $n = 5$ et $f = f(0)\ f(1)\ f(2)\ f(3)\ f(4) = 0\ 1\ 0\ 3\ 1$.

L'idée de l'algorithme est d'utiliser l'élément $f(i)$ du tableau f pour choisir la valeur $p(i)$ de la permutation p dans la liste l des valeurs non encore choisies. Le tableau f étant factoriel,

```
let create_cursor (n: int) : cursor
  requires { n ≥ 0 }
  ensures { sound result }
  ensures { min_lex result.current }
= let a = make n 0 in
  { current = a; new = true }

let next (c: cursor) : unit
  requires { sound c }
  ensures { sound c }
  ensures { c.new →
     inc (old c.current) c.current }
  ensures { not c.new →
     max_lex c.current }
= let a = c.current in
  let n = a.length in
  'L: let r = ref (n-1) in
  while !r ≥ 0 && a[!r] = !r do
    invariant { -1 ≤ !r ≤ n-1 }
    invariant { is_id_sub a (!r+1) n }
    variant  { !r + 1 }
    r := !r - 1
  done;
  if (!r < 0) then (* Last array reached. *)
    c.new ← false
  else begin
    a[!r] ← a[!r] + 1;
    for i = !r+1 to n-1 do
      invariant { !r+1 ≤ i ≤ n }
      invariant { (at a 'L)[!r]+1 = a[!r] }
      invariant { cte_sub a (!r+1) i 0 }
      invariant { lt_lex_at (at a 'L) a !r }
      a[i] ← 0
    done;
    c.new ← true
  end
```

Listing 5: Énumération des factoriels en WhyML.

1. Initialiser une variable auxiliaire l avec la liste $[0, 1, \ldots, n-1]$ des n premiers entiers naturels dans l'ordre croissant.
2. Pour i décroissant de $n-1$ à 0, calculer $j = f(i)$, $k = l(j)$, puis stocker k dans $p(i)$ et supprimer k dans l.

(a) Algorithme du code de permutation

				p				
l	i	j	k	0	1	2	3	4
$[0,1,2,3,4]$	4	1	1	_	_	_	_	1
$[0,2,3,4]$	3	3	4	_	_	_	4	1
$[0,2,3]$	2	0	0	_	_	0	4	1
$[2,3]$	1	1	3	_	3	0	4	1
$[2]$	0	0	2	2	3	0	4	1

(b) Exemple d'exécution

Figure 1: Algorithme et exemple de trace d'exécution

son élément $f(i)$ désigne toujours un élément dans la liste, même si la taille de la liste est décrémentée à chaque itération, par suppression de la valeur choisie.

Le listing 6 présente une fonction `code` qui implémente l'algorithme en WhyML, avec l'aide de quelques fonctions auxiliaires. Pour $n \geq 0$, l'expression `(id_aux n i)` construit la liste $[i, i+1, \ldots, i+n-1]$ (vide si $n \leq 0$). Comme le spécifie le contrat formel de la fonction `id`, l'expression `(id n)` construit la liste $[0, 1, \ldots, n-1]$ pour tout entier $n \geq 0$. Sa deuxième postcondition est exprimée à l'aide de la fonction `nth` d'accès au `n`-ème élément d'une liste `l`, spécifiée par

```
function nth (n: int) (l: list 'a) : 'a
axiom nth_cons_0: ∀ x:'a, r:list 'a. nth 0 (Cons x r) = x
axiom nth_cons_n: ∀ x:'a, r:list 'a, n:int. n > 0 → nth n (Cons x r) = nth (n-1) r
```

dans la bibliothèque standard de Why3. Après ajout d'un contrat à la fonction `id_aux` – non détaillé car élémentaire – le contrat de la fonction `id` est démontré avec Alt-Ergo 1.30.

Comme le spécifie le contrat de la fonction `rm_nth`, l'exécution de l'expression `(rm_nth x l)`

```
let rec id_aux (n i: int) : list int =
  if n ≤ 0 then Nil else
    Cons i (id_aux (n-1) (i+1))

predicate is_id (l:list int) = ∀ i:int.
  0 ≤ i < L.length l → nth i l = i

let id (n: int) : list int
  requires { 0 ≤ n }
  ensures { L.length result = n }
  ensures { is_id result }
= id_aux n 0

let rec rm_nth (x:int)
  (l:ref (list int)) : int
  requires { 0 ≤ x < L.length !l }
  ensures { result = nth x (old !l) }
  ensures { ∀ i. 0 ≤ i < x
    → nth i !l = nth i (old !l) }
  ensures { ∀ i. x < i < L.length !l
    → nth i !l = nth (i+1) (old !l) }
  ensures { length !l =
              length (old !l) - 1 }
  variant { L.length !l }
= match (!l) with
  | Nil → l := Nil; 0
  | Cons y m →
      if x = 0 then begin
        l := m; y
      end else begin
        let r = ref m in
        let z = rm_nth (x-1) r in
        l := Cons y !r;
        z
      end
  end

let code (f: array int) : array int
  requires { is_fact f }
  ensures { is_permut result }
= let n = f.length in
  let p = make n 0 in
  let l = ref (id n) in
  for i = n-1 downto 0 do
    let j = f[i] in
    let k = rm_nth j l in
    p[i] ← k
  done;
  p
```

Listing 6: Code de permutation en WhyML.

a deux effets : (1) elle retourne le $(x+1)$-ème élément de la liste $!l$ référencée par l, avant modification de cette liste par la fonction (première postcondition), et (2) elle supprime cet élément dans cette liste $!l$ (deuxième, troisième et quatrième postconditions). La fonction n'assure ses postconditions que lorsque la position x existe dans la liste, comme spécifié dans sa précondition. Sous cette condition, la longueur de la liste $!l$ est un variant de la fonction `rm_nth`, puisqu'elle est décrémentée à chaque appel récursif de la fonction.

4.4 Preuve de correction du code de permutation

Nous démontrons formellement que la fonction `code` respecte son contrat, qui formalise la conjecture suivante : Si le paramètre f est un tableau factoriel (précondition), la fonction retourne (le tableau de valeurs d')une permutation (postcondition).

Les permutations sont caractérisées 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`.

Les invariants suivants pour la boucle de la fonction `code` permettent la vérification automatique de sa correction :

```
1  invariant { i = L.length !l - 1 }
2  invariant { is_blist !l n }
3  invariant { injlist !l }
```

```
4  invariant { range_sub p (i+1) n n }
5  invariant { inj_sub p (i+1) n }
6  invariant { disj_sub p (i+1) n !l }
```

Le premier invariant déclare que la longueur de la liste $!l$ est toujours égale à $i+1$. C'est le cas puisque chaque itération décrémente i et enlève un élément dans cette liste. Nous expliquons les autres invariants d'abord globalement, puis un par un, en justifiant leur pertinence et leur nécessité.

En général, les invariants de boucle sont introduits pour automatiser la démonstration qu'une certaine propriété est satisfaite après exécution de la boucle. Il s'agit ici de démontrer la propriété (`is_permut p`) que le tableau p est une permutation. C'est le cas puisque, d'une part, chaque itération préserve la propriété que le sous-tableau $p[i+1..n-1]$ est une permutation, et, d'autre part, l'indice i vaut -1 après la dernière itération de la boucle.

La caractérisation `is_permut` d'une permutation comme endofonction sur $[0..n-1]$ (`range p`) et injection (`injective p`) permet de raisonner indépendamment sur ces deux propriétés, d'où les invariants 4 et 5, qui affirment respectivement que le sous-tableau $p[i+1..n-1]$ est à valeurs dans $[0..n-1]$ et injectif (sans doubles).

L'invariant 4 ($p[i+1..n-1]$ à valeurs dans $[0..n-1]$) est préservé car la valeur k stockée dans $p[i]$ à chaque itération vient d'une liste d'entiers dans $[0..n-1]$, comme spécifié par l'invariant 2 (le prédicat `is_blist` n'est pas détaillé car c'est seulement l'analogue pour une liste du prédicat `is_barray` présenté dans la partie 2.2).

Enfin, l'invariant 5 ($p[i+1..n-1]$ sans doubles) découle de l'invariant 6, selon lequel les contenus du tableau $p[i+1..n-1]$ et de la liste $!l$ sont disjoints, et de l'invariant 3, selon lequel la liste $!l$ est sans doubles.

Ainsi, ces annotations de boucle ne sont que la formalisation de propriétés intermédiaires qu'on établirait naturellement lors d'un raisonnement informel sur la correction de ce programme. Why3 et le prouveur Alt-Ergo suffisent pour vérifier toutes ces annotations et donc prouver formellement la conjecture initiale.

5 Conclusion

Le domaine de la combinatoire énumérative est encore peu perméable aux pratiques du génie logiciel. La recherche en combinatoire énumérative est essentiellement une activité de réflexion humaine et de démonstration sur papier, demandant beaucoup d'expérience et d'intuition. Nous pensons que cette activité délicate peut être facilitée par une formalisation machine des problèmes dès le début de leur étude, accompagnée d'une utilisation systématique d'outils logiciels et de méthodes formelles de spécification et de vérification.

Nous désignons par "combinatoire formelle" l'étude de la combinatoire avec un ordinateur. Ceci inclut l'usage très répandu d'un système de calcul formel, mais aussi la programmation d'algorithmes de génération exhaustive bornée ou aléatoire d'objets combinatoires, et l'utilisation d'un prouveur automatique ou d'un assistant de preuve pour démontrer avec un ordinateur des théorèmes combinatoires ou des propriétés de programmes combinatoires.

Cet article contribue à la popularisation de la combinatoire formelle, en présentant des outils et activités de preuve formelle de programmes d'énumération et de test de conjectures combinatoires. Plusieurs générateurs exhaustifs bornés ont été implémentés et spécifiés en WhyML. Leur correction et leur complétude ont été prouvées, avec des prouveurs automatiques et un assistant de preuve. Nous avons ainsi complété des vérifications précédentes [22, 23] avec des preuves de complétude. Dans une étude de cas originale, nous avons démontré une propriété d'un code de permutation, qui est une bijection entre nombres factoriels et permutations de

même taille. C'est un travail préliminaire pour la démonstration d'autres propriétés de ce code et l'étude formelle de divers codes de permutation.

Remerciements. Merci aux relecteurs anonymes pour leurs remarques et suggestions.

Références

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

[2] J.-L. Baril, R. Genestier, A. Giorgetti, and A. Petrossian. Rooted planar maps modulo some patterns. *Discrete Mathematics*, 339(4):1199–1205, 2016.

[3] J.-L. Baril and V. Vajnovszki. A permutation code preserving a double Eulerian bistatistic. *Discrete Applied Mathematics*, 224:9–15, 2017.

[4] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *SEFM'04*, pages 230–239. IEEE Computer Society, 2004.

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

[6] J. Bowles and M. B. Caminati. A verified algorithm enumerating event structures. In *CICM'17*, volume 10383 of *LNCS*, pages 239–254. Springer, 2017.

[7] M. Carlier, C. Dubois, and A. Gotlieb. Constraint reasoning in FOCALTEST. In *Proceedings of the 5th International Conference on Software and Data Technologies - Volume 2: ICSOFT*, pages 82–91. SciTePress, 2010.

[8] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279. ACM, New York, 2000.

[9] M. Clochard, J.-C. Filliâtre, C. Marché, and A. Paskevich. Formalizing semantics with an automatic program verifier. In *VSTTE'14*, volume 8471 of *LNCS*, pages 37–51. Springer, 2014.

[10] The Coq Development Team. The Coq Proof Assistant Reference Manual. http://coq.inria.fr/, 2017. Version 8.7.

[11] S. Cruanes. Satisfiability modulo bounded checking. In *Automated Deduction – CADE 26*, volume 10395 of *LNCS*, pages 114–129. Springer, 2017.

[12] The CVC4 SMT solver. http://cvc4.cs.stanford.edu/web/, 2018.

[13] L. De Moura and N. Björner. Z3: An efficient SMT solver. In *TACAS'08*, volume 4963 of *LNCS*, pages 337–340. Springer, 2008.

[14] C. Dubois and A. Giorgetti. Tests and proofs for custom data generators. *Formal Aspects of Computing*, 2018. https://doi.org/10.1007/s00165-018-0459-1.

[15] C. Dubois, A. Giorgetti, and R. Genestier. Tests and proofs for enumerative combinatorics. In *TAP'16*, volume 6792 of *LNCS*, pages 57–75. Springer International Publishing, 2016.

[16] D. Dumont and G. Viennot. A combinatorial interpretation of the Seidel generation of Genocchi numbers. In *Combinatorial Mathematics, Optimal Designs and Their Applications*, volume 6 of *Annals of Discrete Mathematics*, pages 77 – 87. Elsevier, 1980.

[17] P. Dybjer, Q. Haiyan, and M. Takeyama. Combining testing and proving in dependent type theory. In *TPHOLs'03*, volume 2758 of *LNCS*, pages 188–203. Springer, 2003.

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

[19] 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.

[20] R. Genestier. *Vérification formelle de programmes de génération de données structurées.* PhD thesis, Université de Franche-Comté, 2016.

[21] R. Genestier, A. Giorgetti, and G. Petiot. Gagnez sur tous les tableaux. In *JFLA'15*, 2015. https://hal.inria.fr/hal-01099135.

[22] 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.

[23] A. Giorgetti and R. Lazarini. Preuve de programmes d'énumération avec Why3. In *AFADL'18*, pages 14–19, 2018. http://afadl2018.ls2n.fr/wp-content/uploads/sites/38/2018/06/AFADL_Procs_2018.pdf.

[24] A. Giorgetti and V. Senni. Specification and Validation of Algorithms Generating Planar Lehman Words. GASCom'12, https://hal.inria.fr/hal-00753008, 2012.

[25] G. Gonthier. The four colour theorem: Engineering of a formal proof. In *ASCM'07*, volume 5081 of *LNCS (LNAI)*, pages 333–333. Springer, 2008.

[26] G. Gonthier, A. Asperti, J. Avigad, Y. Bertot, C. Cohen, F. Garillot, S. Le Roux, A. Mahboubi, R. O'Connor, S. Ould Biha, I. Pasca, L. Rideau, A. Solovyev, E. Tassi, and L. Théry. A machine-checked proof of the odd order theorem. In *ITP'13*, pages 163–179. Springer, 2013.

[27] D. Jackson and C. Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *IEEE Trans. Softw. Eng.*, 22(7):484–495, 1996.

[28] C.-A. Laisant. Sur la numération factorielle, application aux permutations. In *Bulletin de la S. M. F., tome 16*, pages 176–183, 1888. http://www.numdam.org/item?id=BSMF_1888__16__176_0.

[29] L. Lampropoulos, D. Gallois-Wong, C. Hriţcu, J. Hughes, B. C. Pierce, and L. Xia. Beginner's luck: a language for property-based generators. In *POPL'17*, pages 114–129. ACM, 2017.

[30] D. H. Lehmer. Teaching combinatorial tricks to a computer. In *Proc. Sympos. Appl. Math. Combinatorial Analysis*, volume 10, pages 179–193. Amer. Math. Soc., 1960.

[31] R. Mantaci and F. Rakotondrajao. A permutations representation that knows what "Eulerian" means. *Discrete Mathematics and Theoretical Computer Science*, 4(2):101–108, 2001.

[32] S. Owre. Random testing in PVS. Workshop on Automated Formal Methods (AFM), 2006. http://fm.csl.sri.com/AFM06/papers/5-Owre.pdf.

[33] 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.

[34] V. Vajnovszki. Lehmer code transforms and Mahonian statistics on permutations. *Discrete Mathematics*, 313(5):581 – 589, 2013.