femto-st SCIENCES & TECHNOLOGIES

INSTITUT FEMTO-ST

UMR CNRS 6174

Formalisation et vérification de théories de permutations

Version 1

Alain Giorgetti

Rapport de recherche n° RR-FEMTO-ST-1715

DÉPARTEMENT DISC – 25 novembre 2020

cnrs

UBFC UNIVERSITÉ BOURGOGNE FRANCHE-COMTÉ

UNIVERSITÉ DE FRANCHE-COMTÉ

ensmm École Nationale Supérieure de Mécanique et des Microtechniques

utbm université de technologie Belfort-Montbéliard

Formalisation et vérification de théories de permutations

Version 1

Alain Giorgetti

Département DISC

VESONTIO

Rapport de recherche no RR-FEMTO-ST-1715 25 novembre 2020 (17 pages)

Résumé : Les permutations sont des objets mathématiques fondamentaux, abondamment étudiés, notamment en combinatoire. En informatique, une permutation permet par exemple de spécifier qu'un programme qui trie une collection de données préserve son contenu. Notre objectif est d'étudier l'impact de telle ou telle axiomatisation de la notion de permutation sur la démonstration formelle d'un théorème ou d'une propriété de programme donnés, en particulier dans le cadre de la plateforme de vérification déductive Why3, fondée sur la logique classique du premier ordre. Dans ce but, nous formalisons diverses théories de permutations dans le langage WhyML de Why3.

Nous formalisons d'abord deux théories de permutations, vues comme des bijections puis comme des ordres linéaires sur un ensemble fini quelconque, à partir d'une étude mathématique, par Albert, Bouvel et Féray, de leurs modèles finis et de leur expressivité. Nous démontrons formellement avec Why3 quelques propositions sur leurs modèles. Nous associons ensuite deux permutations à tout ordre total strict sur un intervalle d'entiers, de manière calculatoire, puis nous testons que ces deux permutations sont mutuellement inverses, à l'aide d'AutoCheck, un nouvel outil de test aléatoire et énumératif de propriétés OCaml et Why3. Le tout est à la fois un exercice de vérification formelle outillée d'un travail de logique mathématique et une évaluation des fonctionnalités de Why3 et d'AutoCheck qui facilitent ces vérifications.

Mots-clés : permutation, logique du premier ordre, preuve formelle, test de propriétés

FEMTO-ST Institute, DISC research department
UFR Sciences - Route de Gray - F-25030 BESANÇON cedex FRANCE
Tel: (33 3) 81 66 65 15 – Fax: (33 3) 81 66 64 50 – e-mail: fvernill@femto-st.fr

Formalization and Verification of Permutation Theories

Version 1

Abstract: Permutations are fundamental mathematical objects, abundantly studied, especially in combinatorics. In computer science, a permutation can be used to specify that a program that sorts a collection of data preserves its content. Our objective is to study the impact of various axiomatizations of the notion of permutation on the formal proof of a theorem or of a given program property, in particular in the context of the Why3 deductive verification platform, based on classical first-order logic. For this purpose, we formalize various permutation theories in the WhyML language of Why3.

We first formalize finite permutations as bijections, and then as linear orders on any finite set, as in the mathematical study of their models and their expressiveness by Albert, Bouvel and Féray. We formally prove with Why3 some propositions on their models. We also associate two permutations to any strict total order over an interval of integers, in a computational way, and we test that these two permutations are mutually inverse, with AutoCheck, a new random and enumerative testing tool for OCaml and Why3 properties. This work is an exercise of formalization and formal verification of a piece of mathematical logic, but also an evaluation of the features of Why3 and AutoCheck that facilitate these activities.

Key-words: permutation, first-order logic, formal proof, property-based testing

FEMTO-ST Institute, DISC research department
UFR Sciences - Route de Gray - F-25030 BESANÇON cedex FRANCE
Tel: (33 3) 81 66 65 15 – Fax: (33 3) 81 66 64 50 – e-mail: fvernill@femto-st.fr

Formalisation et vérification de théories de permutations

Alain Giorgetti
Institut FEMTO-ST, UMR CNRS 6174
Univ. of Bourgogne Franche-Comté
16 route de Gray, F-25030 Besançon Cedex, France
alain.giorgetti@femto-st.fr

Résumé

Les permutations sont des objets mathématiques fondamentaux, abondamment étudiés, notamment en combinatoire. En informatique, une permutation permet par exemple de spécifier qu'un programme qui trie une collection de données préserve son contenu. Notre objectif est d'étudier l'impact de telle ou telle axiomatisation de la notion de permutation sur la démonstration formelle d'un théorème ou d'une propriété de programme donnés, en particulier dans le cadre de la plateforme de vérification déductive Why3, fondée sur la logique classique du premier ordre. Dans ce but, nous formalisons diverses théories de permutations dans le langage WhyML de Why3.

Nous formalisons d'abord deux théories de permutations, vues comme des bijections puis comme des ordres linéaires sur un ensemble fini quelconque, à partir d'une étude mathématique, par Albert, Bouvel et Féray, de leurs modèles finis et de leur expressivité. Nous démontrons formellement avec Why3 quelques propositions sur leurs modèles. Nous associons ensuite deux permutations à tout ordre total strict sur un intervalle d'entiers, de manière calculatoire, puis nous testons que ces deux permutations sont mutuellement inverses, à l'aide d'AutoCheck, un nouvel outil de test aléatoire et énumératif de propriétés OCaml et Why3. Le tout est à la fois un exercice de vérification formelle outillée d'un travail de logique mathématique et une évaluation des fonctionnalités de Why3 et d'AutoCheck qui facilitent ces vérifications.

1 Introduction

Qu'est-ce qu'une permutation ? La question peut sembler triviale, tant ce concept mathématique est élémentaire. La question devient cruciale dans la perspective qui nous motive, qui est la vérification déductive, assistée par ordinateur, de théorèmes sur les permutations et de propriétés de programmes en lien avec ces objets. En effet, cette entreprise est confrontée aux deux enjeux antagonistes de l'expressivité et de l'automaticité. L'*expressivité* est l'étendue des théorèmes formalisables. Elle dépend de la logique de l'outil de preuve utilisé, mais aussi de la théorie choisie pour définir les permutations. Une forte proportion d'*automaticité* dans la synthèse des démonstrations formelles est un critère déterminant pour le choix d'un outil de preuve formelle. Or ces deux enjeux sont contradictoires : plus une théorie est expressive, plus les raisonnements selon cette théorie sont complexes, donc moins ils sont susceptibles d'être découverts par les procédures de décision et les heuristiques implémentées dans les prouveurs automatiques.

Très récemment, Michael Albert, Mathilde Bouvel et Valentin Féray ont étudié l'expressivité de deux théories des permutations sur un ensemble fini, dans le cadre de la logique du premier ordre [ABF20]. Leur théorie TOOB (T*heory* O*f* O*ne* B*ijection*) définit une permutation comme une bijection, tandis que leur théorie TOTO (T*heory* O*f* T*wo* O*rders*) définit une permutation p

de taille n comme un couple $(<_P, <_V)$ d'ordres totaux stricts sur le même ensemble $\{e_1, \ldots, e_n\}$ de n éléments, appelés *positions* (resp. *valeurs*) lorsqu'ils sont comparés avec $<_P$ (resp. $<_V$). L'ordre $<_P$ sur les positions est défini par $e_1 <_P \ldots <_P e_n$. La correspondance entre ces deux théories est que l'ordre $<_V$ sur les valeurs est tel que $p(e_1) <_V \ldots <_V p(e_n)$. Cette définition correspond à une représentation usuelle des permutations

$$\begin{array}{ccc} e_1 & \ldots & e_n \\ p(e_1) & \ldots & p(e_n) \end{array}$$

appelée *two-line notation*, mais aussi à la *one-line notation* qui représente p seulement par le mot $p(e_1) \ldots p(e_n)$, quand il existe un ordre naturel sur les positions, comme l'ordre sur les entiers pour l'ensemble de positions $[n] = \{1, \ldots, n\}$ [Bó04, page 73]. Réciproquement, tout mot de longueur n sans doublons sur l'alphabet $[n]$ est une représentation formelle d'une permutation sur cet ensemble.

Dans cet article nous présentons d'abord une formalisation d'une partie du contenu mathématique de l'article [ABF20], qui poursuit les objectifs suivants : (1) approfondir l'étude de ce contenu, en particulier en détaillant ses démonstrations, dont les plus élémentaires sont éludées, et (2) utiliser cet exercice pour évaluer l'adéquation d'un environnement de vérification déductive à la formalisation d'un texte mathématique. Ensuite, nous introduisons une restriction de la théorie TOTO dans laquelle l'ordre est décidable. Ceci nous permet de programmer en WhyML des fonctions qui associent à cet ordre deux permutations mutuellement inverses. Leurs propriétés sont vérifiées par une combinaison originale de preuves et de tests.

Pour cet exercice de formalisation nous adoptons la plateforme de vérification déductive Why3 [BFM+20], fondée sur une logique classique du premier ordre avec types. Elle propose le langage WhyML pour la spécification fonctionnelle de programmes impératifs et fonctionnels, avec des annotations formelles (préconditions, postconditions, assertions intermédiaires, invariants et variants de boucle, etc.). Le système utilise toutes ces annotations pour générer des *conditions de vérification*, aussi appelées *buts*, qui sont des formules WhyML. Ensuite, des prouveurs automatiques (comme les solveurs SMT) et des assistants de preuve (comme Coq [Coq20]) peuvent être utilisés pour prouver ces buts. De plus, Why3 propose de nombreuses transformations logiques pour simplifier ces buts, jusqu'à obtenir soit une preuve 100% interactive, soit des sous-buts assez simples pour être déchargés automatiquement. Par exemple, la commande split_vc découpe la vérification d'une fonction en sous-buts correspondant à chacune de ses annotations. Why3 propose quatre *stratégies* de recherche de preuve, nommées Auto level 0 à 3, qui sont des combinaisons de transformations logiques et d'appels aux prouveurs automatiques, qui donnent souvent de bons résultats, dans une limite de temps croissante selon leur numéro. Enfin, Why3 offre aussi un mécanisme d'extraction automatique des programmes vers divers langages cibles, dont OCaml, pour les exécuter.

Peu de connaissances de Why3 sont requises pour lire le code présenté dans cette étude. En effet, ce code est commenté et la formalisation est proche du texte mathématique initial. En complément du manuel de Why3 [BFM+20], deux articles récents détaillent davantage le mécanisme de clonage de modules [Fil20] et la notion de type avec invariant [FP20] utilisés ici.

La partie 2 présente quelques choix et développements initiaux effectués pour placer cette formalisation sur de bons rails. La partie 3 (resp. 4) présente notre formalisation WhyML de la théorie TOOB (resp. TOTO) et de propriétés de ses modèles. La partie 5 développe notre proposition de variante plus concrète de la théorie TOTO. Des conséquences et perspectives de ce travail sont discutées dans la partie 6.

Le code est développé et vérifié avec Why3 1.3.3, Coq 8.9.0, AutoCheck 0.1.1 et les solveurs SMT Alt-Ergo 2.2.0, CVC4 1.6 et Z3 4.7.1. Le code, sa documentation, les démonstrations

formelles et leurs statistiques sont accessibles à partir de la page `https://alaingiorgetti.github.io/autocheck/`.

2 Préparatifs

Comme l'écrasante majorité des exposés mathématiques, le propos de [ABF20] s'inscrit, implicitement, dans le cadre de la théorie des ensembles. Notre formalisation pouvait rester dans ce cadre – la librairie standard de Why3 propose diverses axiomatisations des ensembles – ou choisir de s'inscrire dans le cadre de la théorie des types de Why3. Sur ce point, nous avons privilégié la simplicité et la ré-utilisabilité, au détriment de l'expressivité, en optant pour une formalisation qui voit l'ensemble support d'une permutation comme un type WhyML.

2.1 Relations binaires d'ordre total strict

La signature des théories TOOB et TOTO ne contient aucun symbole fonctionnel. Elle ne contient que des symboles relationnels (binaires). C'est naturel pour les deux ordres de la théorie TOOB, mais c'est plus surprenant pour la bijection de la théorie TOTO, qui aurait pu être représentée par un symbole fonctionnel. Même si ce choix 100% relationnel donne lieu à une axiomatisation inutilement plus lourde pour cette théorie d'une permutation vue comme une bijection, nous l'avons conservé dans notre formalisation, à la fois pour maximiser sa proximité avec le texte mathématique et pour explorer les capacités d'automatisation des raisonnements selon cette axiomatisation plus riche.

```
module EndoRelation
  type t
  predicate rel t t
end

module Transitive
  clone export EndoRelation
  axiom Trans : ∀ x y z:t. rel x y → rel y z → rel x z
end

module Asymmetric
  clone export EndoRelation
  axiom Asymm : ∀ x y:t. rel x y → not rel y x
end

module PartialStrictOrder
  clone export Transitive with axiom Trans
  clone export Asymmetric with type t = t, predicate rel = rel, axiom Asymm
end

module TotalStrictOrder
  clone export PartialStrictOrder with axiom Trans, axiom Asymm
  axiom Trichotomy : ∀ x y:t. rel x y ∨ rel y x ∨ x = y
end
```

Listing 1 – Théories d'endorelations binaires de Why3 utilisées ici.

Pour les relations d'ordre total strict, nous ré-utilisons des axiomes et des modules donnés dans le fichier `relations.mlw` de la librairie standard de Why3, et reproduits dans le listing 1. Ils définissent qu'une endorelation `rel` sur un type `t` est un ordre total strict si elle est transitive, asymétrique et totale, au sens de l'axiome `Trichotomy`. La syntaxe `with .. axiom Ax` de WhyML signifie qu'on admet l'axiome `Ax`.

2.2 Graphe d'une injection, surjection ou bijection

Nous axiomatisons le caractère bijectif d'une relation binaire quelconque selon les définitions mathématiques suivantes. Le *graphe* d'une fonction f de A dans B est l'ensemble $\{(x, f(x)) \mid x \in A\}$ des couples composés d'un élément x du *domaine* A de la fonction et de son *image* $f(x)$ par la fonction. Une relation binaire est dite *fonctionnelle* si elle est le graphe d'une fonction, et *injective*, *surjective* ou *bijective* si cette fonction l'est. Les axiomes et modules du listing 2 caractérisent ces relations binaires.

```
module Relation
  type t
  type u
  predicate rel t u
end

module Determinism (* or PartialFunction *)
  clone export Relation
  axiom Deter : ∀ x:t. ∀ y z:u. rel x y → rel x z → y = z
end

module Functionality
  clone export Determinism with axiom Deter
  axiom Total : ∀ x:t. ∃ y:u. rel x y
end

module Injectivity
  clone export Relation
  axiom Injec : ∀ x y:t. ∀ a:u. rel x a → rel y a → x = y
end

module Surjectivity
  clone export Relation
  axiom Surjec : ∀ y:u. ∃ x:t. rel x y
end

module Bijectivity
  clone export Functionality with axiom Deter, axiom Total
  clone export Injectivity with type t = t, type u = u, predicate rel = rel, axiom Injec
  clone export Surjectivity with type t = t, type u = u, predicate rel = rel, axiom Surjec
end
```

Listing 2 – Théories pour des relations bijectives.

2.3 Un type pour l'intervalle d'entiers $[l..u]$

Cette partie présente un type WhyML, nommé `bint` (pour `b`*ounded* `int`*eger*), que nous avons créé pour formaliser tout intervalle d'entiers relatifs non vide $[l..u] = \{l, \ldots, u\}$, pour tous les entiers relatifs l et u tels que $l \leq u$. Le cas particulier $[1..n]$ est noté $[n]$. Cet ensemble $[n]$ est le domaine et le codomaine des permutations utilisées dans [ABF20]. Plus généralement, nous utiliserons le type `bint` comme domaine et codomaine d'applications bijectives qui serviront de modèles de permutations.

Le listing 3 montre un extrait du module `Interval` qui définit ce type. Les bornes l et u de l'intervalle sont formalisées par les constantes `low` et `up`. Le mot-clé `val` les déclare sans fixer leur valeur et permet de les utiliser pour programmer. L'axiome `Nonempty` garantit que l'intervalle [`low`..`up`] n'est pas vide.

Le type `bint` est défini, dans les lignes 8 à 11, comme un enregistrement à un seul champ, qui encapsule un entier relatif, soumis à la contrainte logique (invariant de type, ligne 11)

d'appartenir à l'intervalle d'entiers [`low`..`up`]. La construction après le mot-clé `by` fournit l'entier `low` comme témoin que le type n'est pas vide, ce que Why3 démontre automatiquement. La ligne 13 donne un nom à ce témoin.

La ligne 15 déclare que la fonction logique `to_int : bint → int` est un plongement injectif de l'intervalle dans le type des entiers relatifs de Why3. Cette méta-déclaration autorise la vérification de type (le *type checker*) de Why3 à remplacer par `(to_int e)` toute expression logique `e` de type `bint` utilisée là où une expression de type `int` est attendue. Ce mécanisme permet d'alléger les formalisations ultérieures [1].

```
1  module Interval
2    use int.Int
3
4    val constant low : int
5    val constant up : int
6    axiom Nonempty : low ≤ up
7
8    type bint = {
9      to_int: int
10   } invariant { low ≤ to_int ≤ up }
11     by { to_int = low }
12
13   let constant low_bint : bint = { to_int = low }
14
15   meta coercion function to_int
16
17   axiom Extensionality : ∀ i j:bint. to_int i = to_int j → i = j
18
19   let (=) (i j: bint) : bool = to_int i = to_int j
20
21   let predicate lt_bint (i j: bint) = to_int i < to_int j
22   clone relations.TotalStrictOrder with type t = bint, predicate rel = lt_bint
23
24   let function of_int (i:int) : bint
25     requires { low ≤ i ≤ up }
26     ensures { result.to_int = i }
27   =
28     { to_int = i }
29 end
```

Listing 3 – Extrait d'une formalisation des intervalles d'entiers.

En WhyML, une égalité notée `=` est prédéfinie pour chaque type. Sur tout type enregistrement elle est dite *extensionnelle*, car elle correspond à l'égalité champ par champ. (W_1) [2] Dans Why3 1.3.3, nous avons constaté que l'égalité prédéfinie `=` n'est pas extensionnelle sur les types avec invariant, comme `bint`. Cette propriété est ajoutée par l'axiome `Extensionality` de la ligne 17. La ligne 19 définit une implémentation de l'égalité sur le type `bint`, à l'aide de l'égalité booléenne sur le type `int` spécifiée par

```
val (=) (x y : int) : bool ensures { result ↔ x = y }
```

dans la librairie standard. Toutes ces égalités sont notées `=` mais peuvent être distinguées par le type de leurs arguments et leur contexte logique ou de programme.

La ligne 21 définit une relation binaire `lt_bint` sur le type `bint`, par restriction de l'ordre strict $<$ sur les entiers relatifs, défini dans la librairie standard de Why3. Le mot-clé `let` définit cette relation non seulement comme un prédicat, mais aussi comme une fonction booléenne de même nom. La ligne 22 instancie le module `TotalStrictOrder` pour déclarer que la relation

1. La notation `to_int e`, ou la notation équivalente `e.to_int`, est conservée quand elle clarifie le code.
2. La signification des étiquettes de la forme $(W_{...})$ dans ce texte apparaitra claire dans la partie 6.

`lt_bint` est un ordre total strict. Ses trois propriétés d'asymétrie, de transitivité et de totalité sont démontrées automatiquement, par exemple par Alt-Ergo. L'axiome d'extensionnalité est utilisé dans le troisième cas `x = y` de la démonstration de la propriété `Trichotomy` de totalité, présentée dans le listing 1.

Enfin, les lignes 24 à 28 du listing 3 définissent un constructeur `of_int` pour le type `bint`, qui ne peut être appliqué qu'à un entier `i` entre `low` et `up` (précondition). Le lemme

```
lemma of_int_quasi_inj :
  ∀ i j: int. low ≤ i ≤ up → low ≤ j ≤ up → of_int i = of_int j → i = j
```

dérive de cette définition que la restriction de la fonction `of_int` à l'intervalle [`low`..`up`] est injective. Le lemme

```
lemma of_intK: ∀ i:bint. of_int (to_int i) = i
```

établit que la fonction `of_int` est un inverse à gauche de la fonction `to_int`.

2.4 Fonctions inverses, injections, surjections et bijections

On a fréquemment besoin de spécifier ou de démontrer qu'une fonction f est un inverse à gauche d'une fonction g, et d'obtenir comme conséquences que g est injective et f est surjective. Si de plus g est un inverse à gauche de f, alors les deux fonctions sont bijectives et mutuellement inverses. Le listing 4 présente le contenu d'un fichier `functions.mlw` qui formalise ces propriétés, à partir de l'axiome `Cancel` qui formalise que `u2t` est un *inverse à gauche* (ou une *rétraction*) de `t2u`. L'ajout `as Right` dans le clone du module `Surjective` permet de distinguer les axiomes de même nom dans tout clone du module `Bijective`.

```
module Injective
  type t
  type u
  function t2u t : u
  function u2t u : t

  axiom Cancel : ∀ x : t. u2t (t2u x) = x
  lemma Injec : ∀ x y : t. t2u x = t2u y → x = y
  lemma Surjec : ∀ x : t. ∃ y : u. u2t y = x
end

module Surjective
  type t
  type u
  function t2u t : u
  function u2t u : t
  clone export Injective with type t = u, type u = t,
    function t2u = u2t, function u2t = t2u, axiom Cancel
end

module Bijective
  clone export Injective with axiom Cancel
  clone Surjective as Right with type t = t, type u = u,
    function t2u = t2u, function u2t = u2t, axiom Cancel
end
```

Listing 4 – Fonctions inverses et leurs conséquences.

Ce fichier joue le même rôle que le fichier `function.mlw` de la librairie standard de Why3 1.3.3 [3], mais il l'améliore sur les points suivants : (1) Le nom "`functions`" du fichier n'étant pas le

3. `https://gitlab.inria.fr/why3/why3/-/blob/1.3.3/stdlib/function.mlw`.

mot-clé `function` de Why3, il peut être utilisé sans guillemets en WhyML. Par exemple, on peut écrire `functions.Injective`, au lieu de `"function".Injective` ; (2) Les `goals` de `function.mlw` sont remplacés par des lemmes, afin d'être disponibles dans tout clone de ces modules ; (3) Les noms des types, fonctions, axiomes et lemmes clarifient mieux leur signification.

3 Une permutation est une bijection

Le code WhyML

```
module TOOB
  type t
  predicate rel t t

  clone export Bijectivity with type t = t, type u = t, predicate rel = rel, axiom .
end
```

formalise la signature de la théorie TOOB, réduite à un seul symbole relationnel binaire, noté R dans [ABF20] et `rel` ici. La notation `axiom .` abrège avantageusement la déclaration `axiom Deter, axiom Total, axiom Injec, axiom Surjec` que ce symbole `rel` représente une bijection.

Les auteurs de [ABF20] ont choisi les fonctions bijectives sur $[n]$ comme modèles de TOOB. Plus généralement, nous leur faisons correspondre les données de type `bint → bint`, en spécialisant le type polymorphe `map`, qui est défini dans la librairie standard par

```
type map 'a 'b = 'a → 'b
```

et qui équipe ces *applications* avec quelques fonctions, notations, prédicats et lemmes. Certains de ces prédicats formalisent la bijectivité des applications sur un intervalle d'entiers relatifs, mais bien sûr pas sur le type `bint` dont nous avons besoin. Nous ajoutons donc les définitions générales suivantes des applications injectives, surjectives et bijectives :

```
predicate injective (m: map 'a 'b) = ∀ i j: 'a. m[i] = m[j] → i = j
predicate surjective (m: map 'a 'b) = ∀ j: 'b. ∃ i: 'a. m[i] = j
predicate bijective (m: map 'a 'b) = injective m ∧ surjective m
```

Ensuite, toute permutation sera définie comme σ dans les lignes suivantes :

```
val constant sigma : map bint bint
axiom Sigma_bij : bijective sigma
```

Le modèle de TOOB associé à la permutation σ de $[n]$ est le couple (A^σ, R^σ) où $A^\sigma = [n]$ et R^σ est tel que $i\ R^\sigma\ j$ si et seulement si $\sigma(i) = j$ [ABF20, partie 2.2]. Cette correspondance entre σ et R^σ est formalisée par le prédicat

```
predicate rel_sigma (i j: bint) = map_rel sigma i j
```

où le prédicat `map_rel` – défini dans la ligne 11 du listing 5 – associe son graphe à toute application. Les auteurs de [ABF20] doivent considérer la proposition suivante et sa démonstration comme évidentes, puisque leur texte ne les mentionne même pas.

Proposition 1 *Le couple (A^σ, R^σ) est un modèle de la théorie TOOB.*

Cette proposition 1 peut être simplement formalisée par l'instanciation

```
clone TOOB with type t = bint, predicate rel = rel_sigma
```

du module `TOOB` avec le type `bint` qui formalise un intervalle $[l..u]$ et le prédicat binaire `rel_sigma` qui formalise la relation R^σ pour toute permutation σ sur $[l..u]$. Ce clonage exige la démonstration des instances correspondantes des quatre axiomes du module `TOOB`. Les axiomes `Deter` et

`Injec` sont automatiquement prouvés, par exemple avec Alt-Ergo. L'axiome `Total` n'est prouvé ni par Alt-Ergo ni par CVC4, mais il l'est par Z3. Enfin, l'axiome `Surjec` n'est prouvé automatiquement par aucune stratégie Auto level .. disponible dans Why3, mais il l'est interactivement, par la transformation `inline_all` suivie par un appel à CVC4 ou à Z3. Compte tenu de cette preuve interactive requise, composée de deux transformations, et d'une preuve automatique qui échappe à 2 solveurs parmi 3, démontrer la proposition 1 n'est pas si élémentaire.

Montrons à présent que tout modèle de TOOB est isomorphe à un modèle de permutation [ABF20, Proposition 2].

Proposition 2 *Pour tout modèle (A, R) de TOOB, il existe une permutation σ telle que (A, R) et (A^σ, R^σ) sont isomorphes.*

Le listing 5 présente une formalisation de cette proposition. Le couple (`a`, `rel`) déclaré dans les lignes 1 et 2 formalise le couple (A, R). La ligne 3 déclare que ce couple est un modèle de (notre formalisation de) la théorie TOOB, puisqu'un modèle doit satisfaire tous les axiomes d'une théorie.

```
1  type a
2  predicate rel a a
3  clone TOOB with type t = a, predicate rel = rel, axiom .
4
5  function a2bint a : bint
6  function bint2a bint : a
7  clone functions.Bijective with type t = a, type u = bint,
8    function t2u = a2bint, function u2t = bint2a, axiom Cancel, axiom Right.Cancel
9
10 predicate map_rel (m: map 'a 'b) (x: 'a) (y: 'b) = (m[x] = y)
11 predicate isomorphic (sigma : map bint bint) =
12   ∃ f: map a bint. bijective f ∧ ∀ x y:a. rel x y ↔ map_rel sigma (f x) (f y)
13
14 lemma ex_rel_permut :
15   ∃ rel_permut : map bint bint. ∀ i j:bint. rel_permut[i] = j ↔ rel (bint2a i) (bint2a j)
16
17 lemma Proposition2 : ∃ sigma: map bint bint. bijective sigma ∧ isomorphic sigma
```

Listing 5 – Formalisation de la proposition 2.

L'article [ABF20] ne porte que sur des modèles finis. Pour spécifier que l'ensemble A est fini, nous déclarons dans les lignes 5 à 8 l'existence d'un couple (`a2bint`, `bint2a`) de bijections mutuellement inverses entre les types `a` et `bint`, par clonage du module `Bijective` du fichier `functions.mlw` présenté dans la partie 2.4.

Deux modèles (A, R) et (A', R') sont *isomorphes* s'il existe une bijection f de A sur A' telle que $x\ R\ y$ si et seulement si $f(x)\ R'\ f(y)$ [ABF20]. Avec les deux prédicats définis dans les lignes 10 à 12, la formule `(isomorphic sigma)` exprime que les modèles (`a`, `rel`) et (`bint`, `map_rel sigma`) sont isomorphes.

La proposition 2 est formalisée dans la ligne 17. La clé de sa démonstration est le lemme `ex_rel_permut`, qui affirme l'existence d'une application dont le graphe est isomorphe à la relation `rel` qui formalise R. Ce lemme est démontré interactivement avec Coq, puis la proposition 2 est démontrée interactivement avec Why3.

4 Une permutation est un couple d'ordres totaux stricts

À l'aide de la théorie d'un ordre total strict présentée dans la partie 2.1, la théorie TOTO peut être formalisée en WhyML comme dans le listing 6. Les prédicats `ltP` et `ltV` formalisent

respectivement les ordres totaux stricts $<_P$ et $<_V$ sur les positions et les valeurs, sur le même type `t`.

```
module TOTO
  type t                (* type for positions and values *)
  predicate ltP t t     (* strict total order on positions *)
  predicate ltV t t     (* strict total order on values *)
  clone relations.TotalStrictOrder with type t = t, predicate rel = ltP, axiom .
  clone relations.TotalStrictOrder as V with type t = t, predicate rel = ltV, axiom .
end
```

Listing 6 – Théorie TOTO en WhyML.

4.1 Un type pour les arcs du graphe d'une fonction

Le modèle de la théorie TOTO associé à la permutation σ sur $[n]$ a pour domaine l'ensemble $A^\sigma = \{(i, \sigma(i)) \mid i \in [n]\}$ [ABF20] des arcs du graphe de la fonction σ. Nous proposons de formaliser A^σ par le type `arrow` défini dans le module `MapGraph` du listing 7. Grâce à l'invariant de type, les habitants du type `arrow` sont les arcs (`i`, `m[i]`) du graphe de l'application `m`, pour `i` entre les bornes `low` et `up` du type `bint`. La clause `by` utilise le nombre `low_bint` (défini dans la partie 2.3) pour construire un témoin que ce type n'est pas vide.

```
1  module MapGraph
2    use int.Int
3    use map.Map
4    use Interval
5
6    val constant m : map bint bint
7    type arrow = {
8      source : bint;
9      target : bint
10   } invariant { target = m[source] }
11     by { source = low_bint; target = m[low_bint] }
12
13   axiom Extensionality: ∀ a b:arrow. a.source = b.source ∧ a.target = b.target → a = b
14
15   let (=) (a b: arrow) : bool = a.source = b.source && a.target = b.target
16
17   let predicate lt_source (a b: arrow) = lt_bint a.source b.source
18   clone export relations.TotalStrictOrder with type t = arrow, predicate rel = lt_source
19
20   lemma Source_inj: ∀ a b:arrow. a.source = b.source → a = b
21 end
```

Listing 7 – Type pour les arcs du graphe d'une fonction.

Comme pour le type `bint` (cf. partie 2.3), un axiome déclare que l'égalité prédéfinie `=` est extensionnelle (ligne 13) et une implémentation de l'égalité sur le type `arrow` est définie (ligne 15). La ligne 17 définit simultanément une relation binaire et une fonction booléenne `lt_source` sur le type `arrow`, puis la ligne 18 déclare que cette relation est un ordre total strict. Comme pour le type `bint`, la preuve de la propriété `Trichotomy` de totalité de cet ordre utilise l'axiome `Extensionality` d'extensionnalité de l'égalité. Enfin, le lemme `Source_inj` (ligne 20) formalise que la valeur du champ `target` est déterminée par celle du champ `source`.

4.2 Modèle associé à une permutation

Le modèle de la théorie TOTO associé à la permutation σ sur $[n]$ est le triplet $(A^\sigma, <_P^\sigma, <_V^\sigma)$ tel que $<_P^\sigma$ (resp. $<_V^\sigma$) est l'ordre induit sur la première (resp. seconde) composante des éléments

de A^σ par l'ordre naturel strict $<$ sur les naturels [ABF20]. Autrement dit, $<_P^\sigma$ est la restriction de $<$ sur $[n]$, fermeture transitive de la relation $\{(i, i+1) \mid 1 \leq i < n\}$, et $<_V^\sigma$ est l'unique ordre total strict tel que $\sigma(1) <_V^\sigma \ldots <_V^\sigma \sigma(n)$. C'est la fermeture transitive de la relation $\{(\sigma(i), \sigma(i+1)) \mid 1 \leq i < n\}$.

Proposition 3 *Le triplet $(A^\sigma, <_P^\sigma, <_V^\sigma)$ est un modèle de la théorie* TOTO.

Le listing suivant formalise ces définitions et cette proposition, mathématiquement si évidente que les auteurs de [ABF20] n'en parlent même pas.

```
constant sigma : map bint bint
axiom sigma_bij : bijective sigma
clone MapGraph with constant m = sigma, axiom Source_inj
let predicate ltP_sigma (a b: arrow) = lt_bint a.source b.source
let predicate ltV_sigma (a b: arrow) = lt_bint a.target b.target
clone TOTO with type t = arrow, predicate ltP = ltP_sigma, predicate ltV = ltV_sigma
```

Les trois axiomes d'un ordre total strict sont démontrés automatiquement (par exemple par Alt-Ergo), pour les prédicats `ltP_sigma` et `ltV_sigma` qui formalisent respectivement $<_P^\sigma$ et $<_V^\sigma$.

4.3 Permutation associée à deux ordres totaux stricts

Un analogue à la proposition 2 serait la proposition suivante.

Proposition 4 *Pour tout modèle $(A, <_P, <_V)$ de* TOTO, *il existe une permutation σ telle que $(A, <_P, <_V)$ et $(A^\sigma, <_P^\sigma, <_V^\sigma)$ sont isomorphes.*

Cette proposition est considérée comme si élémentaire dans [ABF20] qu'elle n'y est pas explicitée sous la forme d'une proposition numérotée. Elle est seulement justifiée par un paragraphe (de 6 lignes) qui associe à tout ordre total strict $<$ sur A une fonction *rank* sur A telle que *rank*(a) est le nombre d'éléments de A inférieurs à a selon $<$, augmenté de 1 pour que le codomaine de *rank* soit $[n]$.

Formaliser et caractériser une telle fonction pour tout type WhyML est un travail conséquent. Par ailleurs, la librairie standard de Why3 fournit une fonction

```
numof (p: int → bool) (a b: int) : int
```

pour compter le nombre d'entiers qui satisfont un prédicat `p` donné sur un intervalle `[a..b-1]` donné, et 11 lemmes pour ses propriétés. Afin de ré-utiliser cette fonction et ces lemmes, et d'alléger notre travail de formalisation et sa présentation, nous avons choisi de renoncer à l'objectif de démontrer formellement la proposition 4 dans toute sa généralité, et de nous focaliser sur les modèles où A est un intervalle d'entiers. Dans ce cas, l'ordre $<_P$ peut être identifié à l'ordre naturel sur les entiers, de sorte qu'un seul ordre total strict suffit à modéliser une permutation. Cette approche n'étant pas traitée dans [ABF20], nous la développons dans la partie originale suivante.

5 Une permutation est un ordre total strict

Dans cette partie, nous abandonnons les objectifs de fidélité à [ABF20] et d'universalité de la logique, dont les modèles ont pour domaine un ensemble quelconque, pour nous limiter aux intervalles d'entiers comme supports des ordres totaux stricts et des permutations.

Par contraste avec l'approche logique, descriptive, adoptée jusqu'ici, nous suivons désormais une démarche calculatoire d'implémentation de toutes les fonctions (et de certains prédicats, sous forme de fonctions booléennes). Ceci permet de valider des lemmes par le test, comme illustré dans la partie 5.5, avant d'envisager leur démonstration formelle.

5.1 Théorie d'un ordre total strict

Nous désignons par STOI (pour S*trict* T*otal* O*rder on one integer* I*nterval*) la théorie d'un ordre total strict sur un intervalle d'entiers.

(W_2) Idéalement, nous voudrions imposer à cet ordre strict un type `bint`, comme dans le code suivant :

```
module STOI
  clone export Interval
  val predicate lt bint bint
  clone export relations.TotalStrictOrder with type t = bint, predicate rel = lt, axiom .
end
```

Dans ce code, le module `Interval` est cloné afin que tout clone du module `STOI` – i.e. tout modèle de cette théorie – puisse fixer les valeurs de ses bornes `low` et `up`. Mais tout modèle doit aussi instancier le prédicat `lt`, avec un prédicat binaire sur le type `bint` exporté par le module `STOI`. Il faudrait donc que cette exportation précède cette instantiation, i.e. que la première ligne du pseudo-code suivant ait pour sémantique de fournir le type `bint` requis pour typer `i` et `j` dans la seconde ligne.

```
  clone STOI with val low = .., val up = ..,
  val lt = fun (i:bint) → fun (j:bint) → i.to_int < j.to_int
```

Malheureusement, ce code comporte plusieurs erreurs. La plus bloquante est qu'une instantiation ne peut pas être un terme quelconque, elle ne peut être qu'un nom. Ici, on ne peut pas nommer l'instance de `lt` avant le clonage, car le type qu'il fournit n'existe pas avant ces deux lignes.

Pour contourner cette limitation de WhyML, le module `STOI` est défini pour tout type `t` et le soin de n'instancier ce type que par un type `bint` est laissé à l'utilisateur.

```
module STOI
  type t
  val predicate lt t t
  clone export relations.TotalStrictOrder with type t = t, predicate rel = lt, axiom .
end
```

5.2 Modèle associé à une permutation

Le modèle de la théorie STOI associé à la permutation σ sur $[n]$ est le couple $([n], <^\sigma)$ tel que $i <^\sigma j$ si et seulement si $\sigma(i) < \sigma(j)$.

Proposition 5 *Le couple* $([n], <^\sigma)$ *est un modèle de la théorie STOI.*

Les six lignes suivantes suffisent pour formaliser cette définition et cette proposition.

```
1  val constant size : int
2  clone export Interval with val low = one, val up = size, axiom .
3  val constant sigma : map bint bint
4  axiom sigma_bij : bijective sigma
5  let predicate lt_sigma (i j: bint) = lt_bint (sigma i) (sigma j)
6  clone export STOI with type t = bint, val lt = lt_sigma
```

Les lignes 1 et 2 instancient le type `bint` avec la constante `one` définie dans la librairie standard (par `let constant one : int = 1`), et avec la constante `size`, pour que ce type formalise l'intervalle [`size`]. Par la syntaxe `axiom .`, la contrainte $1 \leq$ `size` est admise. Les lignes 3 et 4 déclarent une permutation σ quelconque sur [`size`]. Sa relation $<^\sigma$ est nommée `lt_sigma` dans la ligne 5. La ligne 6 déclare que ([`size`], `lt_sigma`) est un modèle de la théorie `STOI`. Les trois axiomes d'un ordre total strict sont démontrés automatiquement.

5.3 Permutation associée à un ordre total strict sur un intervalle d'entiers

L'objectif de cette partie est de démontrer la proposition suivante.

Proposition 6 *Pour tout ordre total strict $<_V$ sur un intervalle d'entiers A, il existe une permutation σ sur $[n]$ telle que $(A, <_V)$ et $([n], <_\sigma)$ sont isomorphes.*

Nous construisons deux fonctions candidates pour cette permutation σ. La première implémente la fonction *rank* de [ABF20]. La seconde est une construction originale, présentée dans la partie 5.4. Nous établissons ensuite, dans la partie 5.5, que ces deux fonctions sont mutuellement inverses. L'une ou l'autre de ces deux fonctions peut être appelée *permutation associée à un ordre total strict sur un intervalle d'entiers*.

Pour tout ordre total strict $<$ sur un ensemble fini A, le rang de $a \in A$ est défini par $rank(a) = card\{b \in A \mid b < a\} + 1$ [ABF20, page 9]. Cette fonction est implémentée par la fonction `(rank lt)` du listing 8, pour tout ordre total strict `lt` sur un intervalle d'entiers [`low`..`up`]. Cette définition est compacte car elle ré-utilise la fonction prédéfinie `numof` présentée dans la partie 4.3. Le prédicat exécutable `lt_int` définit à partir de l'ordre `lt` sur `bint` une fonction booléenne `(lt_int lt a : int → bool)` adaptée comme filtre de comptage pour la fonction `numof`.

```
let predicate lt_int (lt : bint → bint → bool) (a:bint) (j:int) =
  low ≤ j ≤ up && lt (of_int j) a

let function rank (lt : bint → bint → bool) (a:bint) : bint
  requires { totalStrictOrder lt }
=
  of_int ((numof (lt_int lt a) low (up+1)) + low)

lemma rank_lt_inj: ∀ lt : bint → bint → bool. totalStrictOrder lt → injective (rank lt)
```

Listing 8 – Définition et injectivité du rang d'un ordre total strict sur un intervalle d'entiers.

La condition de bon typage est que la fonction `rank` soit à valeurs dans l'intervalle [`low`..`up`]. Pour `low`, la preuve est automatique, grâce aux propriétés de `numof` fournies par la librairie standard. Pour montrer que toutes les valeurs de la fonction `rank` sont inférieures ou égales à `up`, nous avons dû ajouter trois lemmes : deux pour `numof` et un pour l'irréflexivité de `lt` : c'est parce que `(lt a a)` est faux que cette borne est `up`, au lieu de `up+1`.

Pour tout ordre total strict `lt`, nous avons démontré avec Coq (en 128 lignes) que la fonction `(rank lt)` est injective (lemme `rank_lt_inj` dans la dernière ligne du listing 8). Pour établir que cette fonction est surjective, nous implémentons une fonction candidate pour son inverse, comme détaillé dans la partie 5.4.

Pour tester un lemme qui concerne tous les ordres totaux stricts, il faut que toutes les fonctions testées soient paramétrées par une représentation de cet ordre, qui est ici la fonction booléenne binaire `lt`. La précondition de la fonction *rank* impose que la fonction `lt` implémente un ordre total strict. Cette précondition limite la ré-utilisabilité de la fonction, mais facilite sa définition et la vérification de ses propriétés. Le prédicat `totalStrictOrder` utilisé est le dernier de la liste de prédicats nommés

```
predicate trans (rel : bint → bint → bool) = ∀ x y z:bint. rel x y → rel y z → rel x z
predicate asymm (rel : bint → bint → bool) = ∀ x y:bint. rel x y → not rel y x
predicate partialStrictOrder (rel : bint → bint → bool) = trans rel ∧ asymm rel
predicate trichotomy (rel : bint → bint → bool) = ∀ x y:bint. rel x y ∨ rel y x ∨ x = y
predicate totalStrictOrder (rel : bint → bint → bool) =
  partialStrictOrder rel ∧ trichotomy rel
```

que nous introduisons selon la même structure que les axiomes et modules du fichier `relations.mlw` de la librairie standard de Why3, présentés dans la partie 2.1.

5.4 Permutation inverse

Soit `lt` un ordre total strict. Cette partie présente une fonction inverse de `(rank lt)`, nommée `(unrank lt)`, implémentée à l'aide de trois fonctions auxiliaires `min`, `max` et `succ`, qui calculent respectivement le minimum d'un ordre total strict, son maximum, et le successeur immédiat selon cet ordre, pour tout nombre autre que son maximum. Ces propriétés caractéristiques de ces trois fonctions auxiliaires sont spécifiées et démontrées formellement. Les implémentations sont impératives (avec une ou deux boucles) pour exploiter et illustrer cet aspect de Why3.

Chaque programme est illustré à l'aide de l'exemple de l'ordre total strict $\prec$ sur [1..6] tel que $4 \prec 6 \prec 2 \prec 5 \prec 1 \prec 3$. Son minimum est $\texttt{min} \prec = 4$. Dans le tableau

Ordre naturel	1	2	3	4	5	6	i	$\texttt{rank}\ j$
Ordre $\prec$	4	6	2	5	1	3	$\texttt{unrank}\ i$	j

la première ligne numérote les colonnes de 1 à 6 et la deuxième ligne contient ces nombres triés dans l'ordre croissant selon $\prec$. Puisque la fonction `(rank` $\prec$`)` associe à chaque nombre son rang selon $\prec$, c'est la fonction qui associe la première ligne à la seconde ligne. Par exemple, $\texttt{rank} \prec 2 = 3$. La fonction inverse `(unrank` $\prec$`)` associe la seconde ligne à la première ligne. Par exemple, $\texttt{unrank} \prec 5 = 1$. La fonction `(succ` $\prec$`)` associe à chaque nombre, excepté le maximum $\texttt{max} \prec = 3$, son successeur de gauche à droite dans la seconde ligne. Par exemple, $\texttt{succ} \prec 6 = 2$.

Le listing 9 propose une implémentation de la fonction `unrank`, qui calcule l'image `j` d'un nombre `i` en itérant `i-1` fois la fonction `succ` à partir du minimum de l'ordre `lt`. Ce programme parcourt la seconde ligne du tableau de gauche à droite jusqu'à atteindre son $\texttt{i}^e$ nombre.

```
let function unrank (lt : bint → bint → bool) (i:bint) : bint
  requires { totalStrictOrder lt }
=
  let ref k = i.to_int in
  let ref j = min lt in
  while not (j = max lt) && k > low do
    variant { k+1-low }
    j := succ lt j;
    k := k-1
  done;
  j
```

Listing 9 – Function `unrank`.

Les implémentations itératives des fonctions `min`, `max` et `succ` sont toutes trois fondées sur un parcours des entiers de `low` à `up` dans l'ordre naturel croissant. Le code des fonctions `min` et `max`, élémentaire, n'est pas présenté.

Le listing 10 présente le contrat et le code de la fonction `succ`. La fonction calcule dans une variable locale `b` le successeur de son paramètre `a`, quand ce dernier n'est pas le maximum de l'ordre total strict `lt` (deuxième précondition, ligne 7 du listing 10). La postcondition (ligne 8) spécifie cette fonctionnalité à l'aide du prédicat `is_succ`, qui formalise que `b` est le successeur de `a` si `a` est inférieur à `b` et s'il n'existe aucun nombre `c` strictement entre `a` et `b`. Sa définition est fondée sur sa généralisation `is_succ_left` à tout sous-intervalle de la forme [`of_int low..of_int i`], utile pour spécifier un invariant de boucle.

La première boucle (lignes 11 à 16) trouve le plus petit entier `i` tel que `a` soit inférieur à `(of_int i)` selon `lt`. Ce nombre est stocké dans la variable `b` (ligne 17). Par exemple, à la ligne 17, `i` vaut 1 lorsque `lt` est l'ordre $\prec$ et `a` vaut 6, car $6 \prec 1$. Puisque `a` n'est pas le maximum de l'ordre `lt`, cet entier `i` existe toujours. La seconde boucle (lignes 19 à 26) parcourt les entiers restants dans l'ordre naturel croissant, afin de stocker dans `b` le plus petit des entiers supérieurs à `a`. Durant cette boucle, `b` contient toujours le plus petit entier supérieur à `a` selon `lt`, parmi les

entiers inférieurs à `i` selon l'ordre naturel (invariant de la ligne 21). Sur l'exemple de l'ordre $\prec$, la seconde boucle parcourt les valeurs de `i` de 2 à 6 tout en donnant la valeur 2 à `b`, car $6 \prec 2 \prec 1$. Les invariants et variants présentés suffisent pour vérifier cette fonction automatiquement, mais seulement avec le plus haut niveau d'automatisation de Why3 (Auto level 3).

```
1  predicate is_succ_left (lt : bint → bint → bool) (a b:bint) (i:int) = lt a b ∧
2    ∀ j. low ≤ j ≤ i → let c = of_int j in not (lt a c && lt c b)
3  predicate is_succ (lt : bint → bint → bool) (a b:bint) = is_succ_left lt a b up
4
5  let succ (lt : bint → bint → bool) (a:bint) : bint
6    requires { totalStrictOrder lt }
7    requires { a ≠ max lt }
8    ensures { is_succ lt a result }
9  =
10   let ref i = low in
11   while i ≤ up && not (lt a (of_int i)) do
12     invariant { low ≤ i ≤ up+1 }
13     invariant { ∀ j. low ≤ j < i → not (lt a (of_int j)) }
14     variant { up - i }
15     i := i+1
16   done;
17   let ref b = of_int i in (* candidate for (succ a) *)
18   i := i+1;
19   while i ≤ up do
20     invariant { low ≤ i ≤ up+1 }
21     invariant { is_succ_left lt a b (i-1) }
22     variant { up - i }
23     let k = of_int i in
24     if lt a k && lt k b then b := k;
25     i := i+1
26   done;
27   b
```

Listing 10 – Function `succ`.

5.5 Test d'un lemme

Les fonctions `(rank lt)` et `(unrank lt)` ont été définies pour être mutuellement inverses, c'est-à-dire pour satisfaire les deux lemmes d'inverse à gauche suivants :

```
lemma   rank_ltK: ∀ lt. totalStrictOrder lt → ∀ i. rank lt (unrank lt i) = i
lemma unrank_ltK: ∀ lt. totalStrictOrder lt → ∀ i. unrank lt (rank lt i) = i
```

Le second lemme se démontre automatiquement avec Why3 ou interactivement avec Coq, à partir du premier lemme et de l'injectivité de `rank lt` (lemme `rank_lt_inj` prouvé avec Coq). La démonstration du premier lemme est laissée en perspective, car c'est une activité potentiellement difficile, voire impossible si les définitions des fonctions `rank` et `unrank` comportent des erreurs. Pour détecter ces erreurs éventuelles, nous présentons ici une démarche de test automatique de ce lemme.

Dans ce but, nous utilisons un prototype d'outil de test automatique de propriétés OCaml et WhyML, nommé AutoCheck [EG20], qui intègre les techniques complémentaires de génération aléatoire et énumérative de données de test. Cet outil exploite le mécanisme d'extraction vers OCaml de Why3, la bibliothèque ENUM de programmes d'énumération spécifiés et implémentés en WhyML et vérifiés avec Why3 [EG19], et l'outil externe QCheck de test aléatoire pour OCaml [CGDM20]. AutoCheck complète ce dernier avec des tests aléatoires pour les propriétés WhyML et des tests énumératifs pour les propriétés WhyML et OCaml.

Le code de test du lemme `rank_ltK` est reproduit dans le listing 11. Il s'agit d'un test énumératif pour tous les ordres totaux stricts sur l'intervalle [1..6]. Les constantes 1 et 6 sont respectivement nommées `one` (dans la librairie standard) et `size` (dans la ligne 6). Dans la ligne 7, le module `Rank`, qui contient les définitions des fonctions testées, est instancié avec ces constantes explicites. Les lignes 1 à 4 importent les composants de l'outil de test.

```
1  use Enum.Permutation
2  use SCheck.Test
3  use SCheck.SCheck
4  use SCheck.SCheck_runner
5
6  let constant size : int = 6
7  clone export Rank with val low = one, val up = size, axiom .
8
9  let function array_sto (a:{array int}) : bint → bint → bool
10   requires { a.length = size ∧ is_permut a }
11   ensures { totalStrictOrder result }
12 =
13   fun x → fun y → a[x.to_int-one] < a[y.to_int-one]
14
15 let function rank_cancel lt : bool
16   requires { totalStrictOrder lt }
17 =
18   for i = one to size do
19     let j = of_int i in
20     if not (rank lt (unrank lt j) = j) then return false
21   done;
22   true
23
24 let rank_cancel_array (a:array int) : bool
25   requires { a.length = size ∧ is_permut a }
26 =
27   rank_cancel (array_sto a)
28
29 let function rank_cancel_test =
30   SCheck_runner.run_tests (Test.make SCheck.(permut_of_size size) rank_cancel_array)
```

Listing 11 – Test énumératif du lemme `rank_ltK`.

Le premier ingrédient d'un test est un générateur des données de test. AutoCheck ne contient pas (encore) de programme d'énumération des ordres totaux stricts sur $[n]$, mais son module `Enum.Permutation` fournit un générateur certifié et efficace de toutes les permutations sur l'ensemble $[0..n-1]$ des n premiers entiers naturels [EG19, EG20]. Il stocke la permutation courante p dans le tableau d'entiers mutable `(a:array int)` de ses images, i.e. $\mathtt{a}[i] = p(i)$ pour $0 \leq i < n$. Nous avons vu dans la partie 5.2 comment associer un ordre total strict à toute permutation. La fonction `array_sto` (lignes 9 à 13) adapte cette correspondance à une permutation sur [0..`size`−1] stockée dans un tableau `a`, pour définir un ordre sur [1..`size`]. Afin que le résultat de la fonction `array_sto` puisse être considéré comme une fonction pure, de type `bint → bint → bool`, la fonction ne doit pas modifier son argument `a`, ce qu'on impose par les accolades autour du type `array int` de `a`. La première précondition de cette fonction garantit que ses lectures dans le tableau se font entre les bornes du tableau. Sa seconde précondition et sa postcondition spécifient sa correction : si le tableau contient les images d'une permutation, alors la fonction retourne un ordre total strict. La démonstration que cette fonction est une bijection est laissée en perspective.

Le second ingrédient d'un test est la fonction booléenne testée, appelée *oracle*. Ici, c'est la fonction `rank_cancel_array` (lignes 24 à 27), qui adapte aux tableaux la fonction booléenne `rank_cancel` (lignes 15 à 22) correspondant au lemme `rank_ltK`.

Le test proprement dit occupe les lignes 29 et 30. La fonction `Test.make` construit le cas de test à partir de l'oracle et d'un générateur énumératif de permutations de taille `size`, encapsulé dans `Scheck.(permut_of_size size)`. Ensuite, la fonction `SCheck_runner.run_tests` exécute le test, en énumérant toutes les données et en évaluant l'oracle pour chaque donnée, tout en comptant le nombre de données évaluées avant échec. Cette validation s'effectue entièrement par exécution du code OCaml produit par extraction automatique du code WhyML. Le verdict du test est soit un contre-exemple, soit le nombre de tests réussis. Pour le présent exemple, ce verdict est :

```
+++ Success ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
Test rank_cancel_array succeeds (ran 720 tests)
```

Ainsi, une confiance est établie dans la correction des fonctions testées, ce qui permet d'envisager avec sérénité la démonstration de ce lemme.

6 Conclusion

Nous avons défini dans le langage WhyML de Why3 trois axiomatisations de la notion de permutation. Les deux premières théories formalisent une partie du contenu logique d'un article de combinatoire récent [ABF20]. La troisième théorie est une restriction calculatoire de la deuxième théorie, créée directement en WhyML par l'auteur. Nous avons utilisé les fonctionnalités de preuve de Why3 et de test d'AutoCheck pour démontrer ou valider diverses propriétés, en maximisant l'automatisation, pour minimiser les efforts et le temps de vérification.

Cette démarche a d'abord permis d'atteindre les deux objectifs annoncés dans l'introduction : (1) un texte mathématique est complété, voire précédé, par un code formel, et (2) des outils logiciels les vérifient avec un haut degré d'automatisation. Ceci montre, par un exemple supplémentaire, la pertinence de la *combinatoire certifiée*, définie comme l'application des méthodes formelles du génie logiciel à la recherche en combinatoire [GM]. On peut étudier certains sujets combinatoires directement de manière formelle, sans trop freiner l'élan créatif, en utilisant les outils de vérification à bon escient.

Par ailleurs, cette étude valorise les outils de vérification utilisés. Certains aspects de Why3 – principalement son mécanisme de clonage de modules et sa notion de type avec invariant – ont montré leur intérêt pour formaliser un exposé sur des théories du premier ordre et leurs modèles, sans avoir recours à l'ordre supérieur. L'étude contribue aussi à l'amélioration des outils, en identifiant des limitations (signalées par (W_1) et (W_2)) et en suggérant des extensions de la librairie standard de Why3 (cf. parties 2.3 et 4.1).

Enfin, l'approche formelle a également un impact non négligeable sur les idées en combinatoire. Par exemple, notre étude utilise un fragment de la librairie standard de Why3 sur les inverses à gauche et leurs conséquences (partie 2.4) pour caractériser un ensemble fini. Ceci suggère d'étudier une théorie de permutations axiomatisant un couple de fonctions mutuellement inverses. Un deuxième exemple est la *one-line notation*, définie dans l'introduction et parfois utilisée comme argument de preuve dans [ABF20]. Pour formaliser cet argument, il faudrait caractériser une permutation sur $[n]$ par une séquence ou une liste de longueur n, à valeurs sans $[n]$ et sans doublons. Ce besoin de formalisation suggère l'étude mathématique de théories où une permutation est un mot sur l'alphabet $[n]$.

Remerciements. Claude Marché, Guillaume Melquiond, Andrei Paskevich et les relecteurs anonymes de la conférence JFLA 2021 sont chaleureusement remerciés pour leurs précieux conseils. Ce travail a été soutenu par l'EIPHI Graduate School (contrat ANR-17-EURE-0002).

Références

[ABF20] Michael Albert, Mathilde Bouvel, and Valentin Féray. Two first-order logics of permutations. *Journal of Combinatorial Theory, Series A*, 171 :105158, April 2020. https://arxiv.org/abs/1808.05459v2.

[BFM+20] François Bobot, Jean-Christophe Filliâtre, Claude Marché, Guillaume Melquiond, and Andrei Paskevich. *The Why3 Platform*, 2020. http://why3.lri.fr/manual.pdf.

[Bó04] Miklós Bóna. *Combinatorics of Permutations*. Discrete mathematics and its applications. Chapman and Hall/CRC, 2004.

[CGDM20] Simon Cruanes, Rudi Grinberg, Jacques-Pascal Deplaix, and Jan Midtgaard. QuickCheck inspired property-based testing for OCaml., 2020. https://github.com/c-cube/qcheck.

[Coq20] The Coq Proof Assistant, 2020. http://coq.inria.fr.

[EG19] Clotilde Erard and Alain Giorgetti. Bounded exhaustive testing with certified and optimized data enumeration programs. In Christophe Gaston, Nikolai Kosmatov, and Pascale Le Gall, editors, *Testing Software and Systems. ICTSS 2019.*, volume 11812 of *LNCS*, pages 159–175, Cham, 2019. Springer.

[EG20] Clotilde Erard and Alain Giorgetti. Random and enumerative testing for OCaml and WhyML properties. https://github.com/alaingiorgetti/autocheck/blob/master/docs/EG20.pdf, soumis, 2020.

[Fil20] Jean-Christophe Filliâtre. Simpler proofs with decentralized invariants, March 2020. https://hal.inria.fr/hal-02518570.

[FP20] Jean-Christophe Filliâtre and Andrei Paskevich. Abstraction and genericity in Why3, June 2020. https://hal.inria.fr/hal-02696246.

[GM] Alain Giorgetti and Nicolas Magaud. Combinatoire certifiée. En cours de publication.

FEMTO-ST INSTITUTE, headquarters
15B Avenue des Montboucons - F-25030 Besançon Cedex France
Tel: (33 3) 63 08 24 00 – e-mail: contact@femto-st.fr

FEMTO-ST — AS2M: TEMIS, 24 rue Alain Savary, F-25000 Besançon France
FEMTO-ST — DISC: UFR Sciences - Route de Gray - F-25030 Besançon cedex France
FEMTO-ST — ENERGIE: Parc Technologique, 2 Av. Jean Moulin, Rue des entrepreneurs, F-90000 Belfort France
FEMTO-ST — MEC'APPLI: 24, chemin de l'épitaphe - F-25000 Besançon France
FEMTO-ST — MN2S: 15B Avenue des Montboucons - F-25030 Besançon cedex France
FEMTO-ST — OPTIQUE: 15B Avenue des Montboucons - F-25030 Besançon cedex France
FEMTO-ST — TEMPS-FREQUENCE: 26, Chemin de l'Epitaphe - F-25030 Besançon cedex France

http://www.femto-st.fr