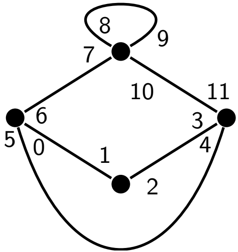
Spécification et vérification formelle d'opérations sur les permutations

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

AFADL 2016

Exemple de carte combinatoire étiquetée



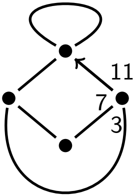
Carte topologique

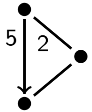
$$
\begin{aligned}
D &= \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\} \\
R &= (5\ 0\ 6)\ (1\ 2)\ (11\ 3\ 4)\ (8\ 7\ 10\ 9) \\
L &= (0\ 1)\ (2\ 3)\ (4\ 5)\ (6\ 7)\ (8\ 9)\ (10\ 11)
\end{aligned}
$$

Carte combinatoire (D, R, L)

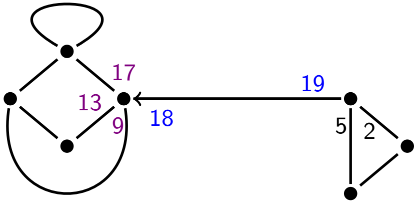
Transitivité: $11 \xrightarrow{L} 10 \xrightarrow{R} 9 \xrightarrow{R} 8 \xrightarrow{R} 7 \xrightarrow{L} 6$

Motivation : construction de cartes





$R_1 = \ldots\ (11\ 7\ 3)\ \ldots$ $R_2 = \ldots\ (5\ 2)\ \ldots$



$R = \ldots\ (18\ 17\ 13\ 9)\ \ldots\ (19\ 5\ 2)\ \ldots$

Raisonnement formel sur les cartes

- Les **cartes** sont des structures combinatoires importantes
- Preuves très techniques
- Davantage d'assistance et de confiance avec des preuves formelles
- Obtenir des algorithmes corrects par construction

Motivations

C. Dubois, A. Giorgetti, and R. Genestier.
Tests and proofs for enumerative combinatorics.
In B. Aichernig and C. A. Furia, editors, *Tests and Proofs (TAP)*, volume 9762 of *LNCS*. Springer, Heidelberg, 2016.

- Formalisation en Coq des notions de permutation et de carte combinatoire
- Deux opérations de construction de cartes combinatoires
  $\rightsquigarrow$ utilisent deux opérations sur les permutations
- Preuves formelles interactives en Coq

Motivations

C. Dubois, A. Giorgetti, and R. Genestier.
Tests and proofs for enumerative combinatorics.
In B. Aichernig and C. A. Furia, editors, *Tests and Proofs (TAP)*, volume 9762 of *LNCS*. Springer, Heidelberg, 2016.

- Formalisation en Coq des notions de permutation et de carte combinatoire
- Deux opérations de construction de cartes combinatoires
  $\rightsquigarrow$ utilisent deux opérations sur les permutations
- Preuves formelles interactives en Coq
- Objectif : Automatisation des preuves de préservation des permutations par ces opérations
- Travail effectué en C + ACSL

Outline

Outline

Prédicats ACSL caractérisant une permutation

Permutation : endofonction injective sur $\{0, \ldots, n-1\}$

Prédicats ACSL caractérisant une permutation

Permutation : endofonction injective sur $\{0, \ldots, n-1\}$

```
/*@ predicate is_fct(int *a, integer b, integer c) =
  @  \forall integer i; 0 <= i < b ==> 0 <= a[i] < c; */
```

Prédicats ACSL caractérisant une permutation

Permutation : endofonction injective sur $\{0, \ldots, n-1\}$

```
/*@ predicate is_fct(int *a, integer b, integer c) =
  @  \forall integer i; 0 <= i < b ==> 0 <= a[i] < c; */

/*@ predicate is_linear(int *a, integer n) =
  @  \forall integer j; 0 <= j < n ==>
  @   \forall integer k; 0 <= k < n ==>
  @    (j != k ==> a[j] != a[k]); */
```

Prédicats ACSL caractérisant une permutation

Permutation : endofonction injective sur $\{0, \ldots, n-1\}$

```
/*@ predicate is_fct(int *a, integer b, integer c) =
  @  \forall integer i; 0 <= i < b ==> 0 <= a[i] < c; */

/*@ predicate is_linear(int *a, integer n) =
  @  \forall integer j; 0 <= j < n ==>
  @   \forall integer k; 0 <= k < n ==>
  @    (j != k ==> a[j] != a[k]); */

/*@ predicate is_perm(int *a, integer n) =
  @  is_fct(a,n,n) && is_linear(a,n); */
```

Insertion

Insertion de n avant i dans p (de longueur n), notée $insert(p, i)$

- Ajoute à p le cycle (n) si $i = n$
- Insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i < n$

Insertion

Insertion de n avant i dans p (de longueur n), notée $insert(p, i)$

- Ajoute à p le cycle (n) si $i = n$
- Insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i < n$

Exemple :
$p = (0\ 1\ 3)\ (2\ 4) = 1\ 3\ 4\ 0\ 2$

Insertion

Insertion de n avant i dans p (de longueur n), notée $insert(p, i)$

- Ajoute à p le cycle (n) si $i = n$
- Insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i < n$

Exemple :
$p = (0\ 1\ 3)\ (2\ 4) = 1\ 3\ 4\ 0\ 2$

- $insert(p, 4) \rightsquigarrow (0\ 1\ 3)\ (2\ 5\ 4) = 1\ 3\ 5\ 0\ 2\ 4$

Insertion

Insertion de n avant i dans p (de longueur n), notée $insert(p, i)$

- Ajoute à p le cycle (n) si $i = n$
- Insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i < n$

Exemple :
$p = (0\ 1\ 3)\ (2\ 4) = 1\ 3\ 4\ 0\ 2$

- $insert(p, 4) \rightsquigarrow (0\ 1\ 3)\ (2\ 5\ 4) = 1\ 3\ 5\ 0\ 2\ 4$
- $insert(p, 5) \rightsquigarrow (0\ 1\ 3)\ (2\ 4)\ (5) = 1\ 3\ 4\ 0\ 2\ 5$

Insertion

Insertion de n avant i dans p (de longueur n), notée $insert(p, i)$

- Ajoute à p le cycle (n) si $i = n$
- Insère l'entier n avant l'entier i dans son cycle dans p si $0 \leq i < n$

Exemple :
$p = (0\ 1\ 3)\ (2\ 4) = 1\ 3\ 4\ 0\ 2$

- $insert(p, 4) \rightsquigarrow (0\ 1\ 3)\ (2\ 5\ 4) = 1\ 3\ 5\ 0\ 2\ 4$
- $insert(p, 5) \rightsquigarrow (0\ 1\ 3)\ (2\ 4)\ (5) = 1\ 3\ 4\ 0\ 2\ 5$

```
void insert(int p[], int i, int q[], int n) {
 if (0 <= i && i <= n) {
  q[n] = i;
  for (int j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
 }
}
```

insert : spécification ACSL

```
/*@ predicate is_loop_insert(int *q, int *p,
  @                 integer b, integer c, integer d) =
  @  \forall integer j; 0 <= j < b ==>
  @   q[j] == ((p[j] == c) ? d : p[j]); */

/*@ predicate is_insert(int *q, int *p, integer b, integer c) =
  @  0 <= b <= c ==> (q[c] == b && is_loop_insert(q,p,c,b,c));

void insert(int p[], int i, int q[], int n) {
 if (0 <= i && i <= n) {
  q[n] = i;
  for (int j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
 }
}
```

Somme directe

Fusionner deux permutations par décalage des valeurs de l'une d'entre elles

Somme directe

Fusionner deux permutations par décalage des valeurs de l'une d'entre elles

Somme directe de p_1 de longueur n_1 et p_2 de longueur n_2, notée $p_1 \oplus p_2$

- $(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$
- $(p_1 \oplus p_2)(i) = p_2(i - n_1) + n_1$ pour $n_1 \leq i < n_1 + n_2$

Somme directe

Fusionner deux permutations par décalage des valeurs de l'une d'entre elles

Somme directe de p_1 de longueur n_1 et p_2 de longueur n_2, notée $p_1 \oplus p_2$

- $(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$
- $(p_1 \oplus p_2)(i) = p_2(i - n_1) + n_1$ pour $n_1 \leq i < n_1 + n_2$

Exemple :

- $p_1 = 2\ 1\ 0 = (0\ 2)\ (1) \in S_3,\ p_2 = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4)$

Somme directe

Fusionner deux permutations par décalage des valeurs de l'une d'entre elles

Somme directe de p_1 de longueur n_1 et p_2 de longueur n_2, notée $p_1 \oplus p_2$

- $(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$
- $(p_1 \oplus p_2)(i) = p_2(i - n_1) + n_1$ pour $n_1 \leq i < n_1 + n_2$

Exemple :

- $p_1 = 2\ 1\ 0 = (0\ 2)\ (1) \in S_3,\ p_2 = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4)$
- $p_1 \oplus p_2 = 2\ 1\ 0\ 4\ 6\ 7\ 3\ 5 = (0\ 2)\ (1)\ (3\ 4\ 6)\ (5\ 7)$

Somme directe

Fusionner deux permutations par décalage des valeurs de l'une d'entre elles

Somme directe de p_1 de longueur n_1 et p_2 de longueur n_2, notée $p_1 \oplus p_2$

- $(p_1 \oplus p_2)(i) = p_1(i)$ pour $0 \leq i < n_1$
- $(p_1 \oplus p_2)(i) = p_2(i - n_1) + n_1$ pour $n_1 \leq i < n_1 + n_2$

Exemple :

- $p_1 = 2\ 1\ 0 = (0\ 2)\ (1) \in S_3,\ p_2 = 1\ 3\ 4\ 0\ 2 = (0\ 1\ 3)\ (2\ 4)$
- $p_1 \oplus p_2 = 2\ 1\ 0\ 4\ 6\ 7\ 3\ 5 = (0\ 2)\ (1)\ (3\ 4\ 6)\ (5\ 7)$

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

Somme directe : spécification ACSL

```
/*@ predicate is_sum(int *p, int *p1, int *p2, integer b, integer c) =
  @  \forall integer i; 0 <= i < b ==>
  @   p[i] == ((i < c) ?  p1[i] : p2[i-c]+c); */

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

Outline

Spécification ACSL de insert

```
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;



    for (j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;

void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;

    for (j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
```

```
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;

  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));



void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;



  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
```

```
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;

  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];


void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;




  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);

void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;




    for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);
  @ ensures is_insert(q,p,i,n); */
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;




    for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);
  @ ensures is_insert(q,p,i,n); */
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;
  /*@ loop invariant 0 <= j <= n;




  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);
  @ ensures is_insert(q,p,i,n); */
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;
  /*@ loop invariant 0 <= j <= n;
    @ loop invariant is_loop_insert(q,p,j,i,n);



  for (j = 0; j < n; j++) q[j] = (p[j] == i) ? n : p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);
  @ ensures is_insert(q,p,i,n); */
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;
  /*@ loop invariant 0 <= j <= n;
    @ loop invariant is_loop_insert(q,p,j,i,n);
    @ loop invariant is_fct(q,j,n+1) && is_linear(q,j);


  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);
  @ ensures is_insert(q,p,i,n); */
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;
  /*@ loop invariant 0 <= j <= n;
    @ loop invariant is_loop_insert(q,p,j,i,n);
    @ loop invariant is_fct(q,j,n+1) && is_linear(q,j);
    @ loop assigns j, q[0..n-1];

  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de insert

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1)) && \valid(q+(0..n));
  @ requires \separated(q+(0..n),p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns q[0..n];
  @ ensures is_perm(q,n+1);
  @ ensures is_insert(q,p,i,n); */
void insert(int p[], int i, int q[], int n) {
  if (0 <= i && i <= n) {
    q[n] = i;
  /*@ loop invariant 0 <= j <= n;
    @ loop invariant is_loop_insert(q,p,j,i,n);
    @ loop invariant is_fct(q,j,n+1) && is_linear(q,j);
    @ loop assigns j, q[0..n-1];
    @ loop variant n-j; */
  for (j = 0; j < n; j++) q[j] = (p[j] == i) ?  n :  p[j];
  }
}
```

Spécification ACSL de `sum`

```
void sum(int p1[], int p2[], int p[], int n1, int n2) {



  for (int i = 0; i < n1+n2; i++)
    p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
}
```

Spécification ACSL de sum

```
/*@ requires n1 >= 0 && n2 >= 0;
  @ requires \valid(p1+(0..n1-1)) && \valid(p2+(0..n2-1));
  @ requires \valid(p+(0..n1+n2-1));
  @ requires \separated(p1+(0..n1-1),p2+(0..n2-1),p+(0..n1+n2-1));
  @ requires is_perm(p1,n1) && is_perm(p2,n2);
  @ assigns p[0..n1+n2-1];
  @ ensures is_perm(p,n1+n2);
  @ ensures is_sum(p,p1,p2,n1+n2,n1); */
void sum(int p1[], int p2[], int p[], int n1, int n2) {





  for (int i = 0; i < n1+n2; i++)
    p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
}
```

Spécification ACSL de sum

```
/*@ requires n1 >= 0 && n2 >= 0;
  @ requires \valid(p1+(0..n1-1)) && \valid(p2+(0..n2-1));
  @ requires \valid(p+(0..n1+n2-1));
  @ requires \separated(p1+(0..n1-1),p2+(0..n2-1),p+(0..n1+n2-1));
  @ requires is_perm(p1,n1) && is_perm(p2,n2);
  @ assigns p[0..n1+n2-1];
  @ ensures is_perm(p,n1+n2);
  @ ensures is_sum(p,p1,p2,n1+n2,n1); */
void sum(int p1[], int p2[], int p[], int n1, int n2) {
  /*@ loop invariant 0 <= i <= n1+n2;
    @ loop invariant is_sum(p,p1,p2,i,n1);
    @ loop invariant is_fct(p,i,n1+n2) && is_linear(p,i);
    @ loop assigns i, p[0..n1+n2-1];
    @ loop variant n1+n2-i; */
  for (int i = 0; i < n1+n2; i++)
    p[i] = (i < n1) ? p1[i] : p2[i-n1]+n1;
}
```

Variante de *insert* : modification du tableau en place

```
void insert_inplace(int p[], int i, int n) {
 if (0 <= i && i <= n) {
  p[n] = i;
  for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
 }
}
```

Variante de *insert* : modification du tableau en place

```
void insert_inplace(int p[], int i, int n) {
 if (0 <= i && i <= n) {
  p[n] = i;
  for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
 }
}
```

`\at(e,L)` ⇝ valeur de l'expression e dans l'état identifié par le label `L`

```
/*@ predicate is_loop_insert_inplace{L1,L2}
  @  (int *p, integer b, integer c, integer d) =
  @  \forall integer j; 0 <= j < \at(b,L2) ==>
  @   \at(p[j],L2) == ((\at(p[j],L1) == c ) ? d : \at(p[j],L1)); */
```

Variante de *insert* : modification du tableau en place

```
void insert_inplace(int p[], int i, int n) {
 if (0 <= i && i <= n) {
  p[n] = i;
  for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
 }
}
```

`\at(e,L)` ⇝ valeur de l'expression e dans l'état identifié par le label `L`

```
/*@ predicate is_loop_insert_inplace{L1,L2}
  @  (int *p, integer b, integer c, integer d) =
  @  \forall integer j; 0 <= j < \at(b,L2) ==>
  @   \at(p[j],L2) == ((\at(p[j],L1) == c ) ? d : \at(p[j],L1)); */

/*@ predicate is_insert_inplace{L1,L2}(int *p, integer b, integer c) =
  @  0 <= b <= c ==> (\at(p[c],L2) == b
  @   && is_loop_insert_inplace{L1,L2}(p,c,b,c)); */
```

Variante de *insert* : modification du tableau en place

```
void insert_inplace(int p[], int i, int n) {
 if (0 <= i && i <= n) {
  p[n] = i;
  for (int j = 0; j < n; j++) if (p[j] == i) p[j] = n;
 }
}
```

`\at(e,L)` ⇝ valeur de l'expression e dans l'état identifié par le label L

```
/*@ predicate is_loop_insert_inplace{L1,L2}
  @  (int *p, integer b, integer c, integer d) =
  @  \forall integer j; 0 <= j < \at(b,L2) ==>
  @   \at(p[j],L2) == ((\at(p[j],L1) == c ) ? d : \at(p[j],L1)); */

/*@ predicate is_insert_inplace{L1,L2}(int *p, integer b, integer c) =
  @  0 <= b <= c ==> (\at(p[c],L2) == b
  @   && is_loop_insert_inplace{L1,L2}(p,c,b,c)); */

/*@ predicate is_eq_gt{L1,L2}(int *p, integer b, integer c) =
  @  \forall integer k; b <= k < c ==>
  @   \at(p[k],L2) == \at(p[k],L1); */
```

Spécification ACSL de insert_inplace

```
void insert_inplace(int p[], int i, int n) {
  if (0 <= i && i <= n) {
    p[n] = i;




    for (j = 0; j < n; j++) if (p[j] == i) p[j] = n;
  }
}
```

Spécification ACSL de insert_inplace

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns p[0..n];
  @ ensures is_perm(p,n+1);

void insert_inplace(int p[], int i, int n) {
  if (0 <= i && i <= n) {
    p[n] = i;
  /*@ loop invariant 0 <= j <= n;


    @ loop invariant is_fct(p,j,n+1) && is_linear(p,j);
    @ loop assigns j, p[0..n-1];
    @ loop variant n-j; */
    for (j = 0; j < n; j++) if (p[j] == i) p[j] = n;
  }
}
```

Spécification ACSL de insert_inplace

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns p[0..n];
  @ ensures is_perm(p,n+1);
  @ ensures is_insert_inplace{Pre,Post}(p,i,n); */
void insert_inplace(int p[], int i, int n) {
  if (0 <= i && i <= n) {
    p[n] = i;
  /*@ loop invariant 0 <= j <= n;


    @ loop invariant is_fct(p,j,n+1) && is_linear(p,j);
    @ loop assigns j, p[0..n-1];
    @ loop variant n-j; */
    for (j = 0; j < n; j++) if (p[j] == i) p[j] = n;
  }
}
```

Spécification ACSL de insert_inplace

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns p[0..n];
  @ ensures is_perm(p,n+1);
  @ ensures is_insert_inplace{Pre,Post}(p,i,n); */
void insert_inplace(int p[], int i, int n) {
  if (0 <= i && i <= n) {
    p[n] = i;
  /*@ loop invariant 0 <= j <= n;
    @ loop invariant is_loop_insert_inplace{Pre,Here}(p,j,i,n);

    @ loop invariant is_fct(p,j,n+1) && is_linear(p,j);
    @ loop assigns j, p[0..n-1];
    @ loop variant n-j; */
    for (j = 0; j < n; j++) if (p[j] == i) p[j] = n;
  }
}
```

Spécification ACSL de insert_inplace

```
/*@ requires 0 <= i <= n;
  @ requires \valid(p+(0..n-1));
  @ requires is_perm(p,n);
  @ assigns p[0..n];
  @ ensures is_perm(p,n+1);
  @ ensures is_insert_inplace{Pre,Post}(p,i,n); */
void insert_inplace(int p[], int i, int n) {
  if (0 <= i && i <= n) {
    p[n] = i;
  /*@ loop invariant 0 <= j <= n;
    @ loop invariant is_loop_insert_inplace{Pre,Here}(p,j,i,n);
    @ loop invariant is_eq_gt{Pre,Here}(p,j,n);
    @ loop invariant is_fct(p,j,n+1) && is_linear(p,j);
    @ loop assigns j, p[0..n-1];
    @ loop variant n-j; */
    for (j = 0; j < n; j++) if (p[j] == i) p[j] = n;
  }
}
```

Outline

Pratique de la démonstration automatique

- Fonctions C utilisant un petit fragment du langage C
- Spécifications exprimées en logique du premier ordre
- Tableaux d'entiers alloués préalablement
- Expressions d'arithmétique linéaire sur les entiers
- Difficultés dans l'établissement des preuves liées à la précision des invariants de boucles
  - Ajouter des assertions
  - Décomposer en sous-fonctions

Conclusion

- Implémentation en C de deux opérations sur les permutations
- Spécification formelle de leur comportement
- Démonstration automatique de leur correction
- 54 obligations de preuve en moins d'une minute, avec Frama-C + WP + Alt-Ergo, CVC3, CVC4
- Durée allouée à chaque solveur étendue à une minute

Conclusion

- Implémentation en C de deux opérations sur les permutations
- Spécification formelle de leur comportement
- Démonstration automatique de leur correction
- 54 obligations de preuve en moins d'une minute, avec Frama-C + WP + Alt-Ergo, CVC3, CVC4
- Durée allouée à chaque solveur étendue à une minute
- D'autres opérations ont été implémentées et prouvées
- Archive `enum.*.tar.gz` disponible à l'adresse `http://members.femto-st.fr/richard-genestier/en`

Questions

- Merci pour votre attention
- Questions ?