Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels

Julien Tesson

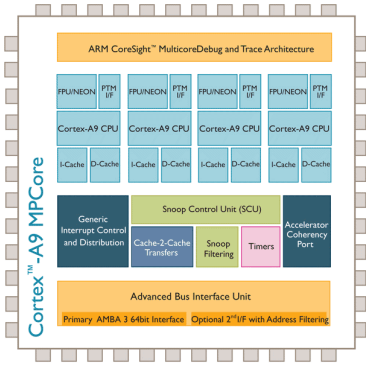
sous la direction de Frédéric Loulergue

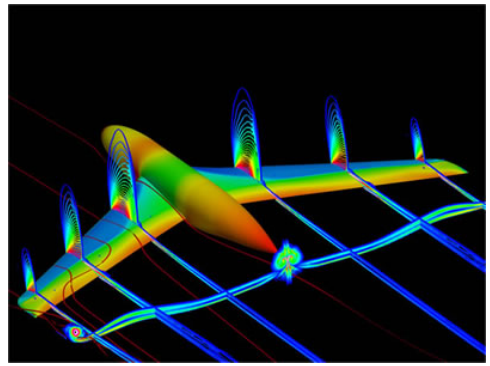
LIFO, Université d'Orléans

Soutenance de Thèse,
8 novembre 2011
Orléans


VISA
VISA

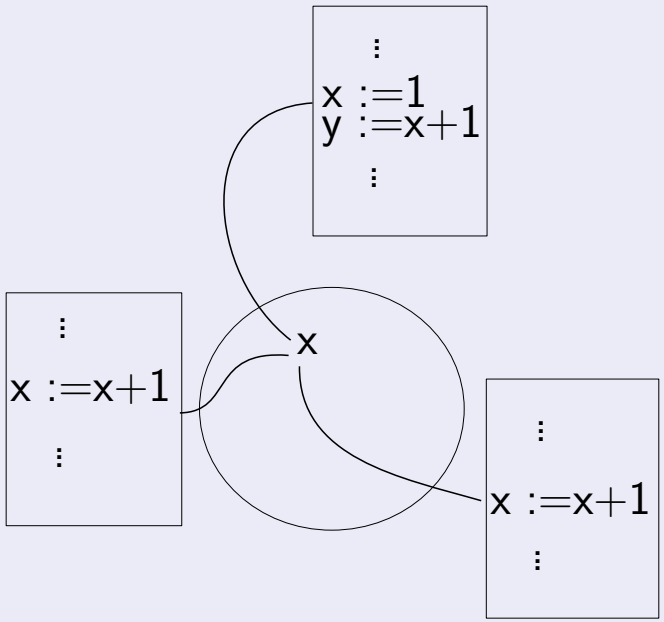

A380
A380 AIRBUS


ARM CoreSight™ MulticoreDebug and Trace Architecture
FPU/NEON
PTM I/F
Cortex-A9 CPU
I-Cache
D-Cache
Cortex™-A9 MPCore
Generic Interrupt Control and Distribution
Snoop Control Unit (SCU)
Cache-2-Cache Transfers
Snoop Filtering
Timers
Accelerator Coherency Port
Advanced Bus Interface Unit
Primary AMBA 3 64bit Interface
Optional 2nd I/F with Address Filtering

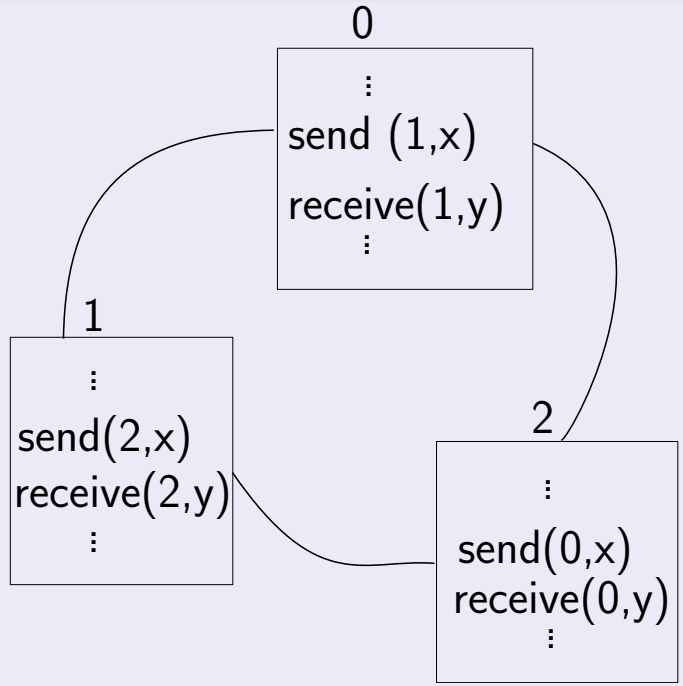


Programmation parallèle

Mémoire partagée



Mémoire distribuée



Programmation parallèle

Modèles bas niveaux

+ Expressif
- Expressif $\longrightarrow$ complexe

Parallélisme structuré

- Squelettes algorithmiques
- BSP : parallélisme quasi-synchrone
- Programmation parallèle fonctionnelle

Squelettes algorithmiques

Origines

- Cole 1989
- Motifs de programmes parallèles

Parallélisme de données

- Structure de données répartie
- Opérations collectives (map)
- Opérations de réduction (scan)
- Opérations de répartition / d'équilibrage

Avantages

Découplage entre modèle de programmation et modèle d'exécution

- *Facile* à utiliser
- Implantations optimisées pour chaque architecture cible

Inconvénient

- Nombre fini de structures de données et d'opérations

Bulk Synchronous Parallelism (BSP)

Origines

Valiant & McColl, années 90

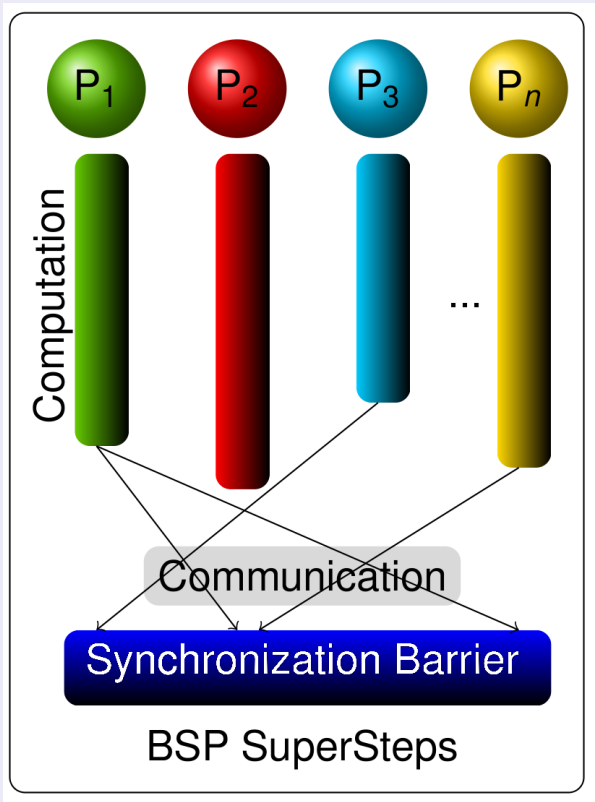
Trois modèles

- Architecture abstraite
- Modèle d'exécution
- Modèle de coût

Calculateur BSP

- p paires mémoire/processeur *(de vitesse r)*
- Un réseaux de communication *(de vitesse g)*
- Une unité de synchronisation *(en temps L)*

Modèle d'exécution



Modèle coût

$$T(s) = \max_{0 \leq i < p} w_i + h \times g + L$$

où $h = \max_{0 \leq i < p} \{h_i^+, h_i^-\}$

Bulk Synchronous Parallel ML

Principes de conception

- Un ensemble de primitives parallèles restreint
- Universel pour le parallélisme BSP
- Vue globale du programme
- Une sémantique formelle simple

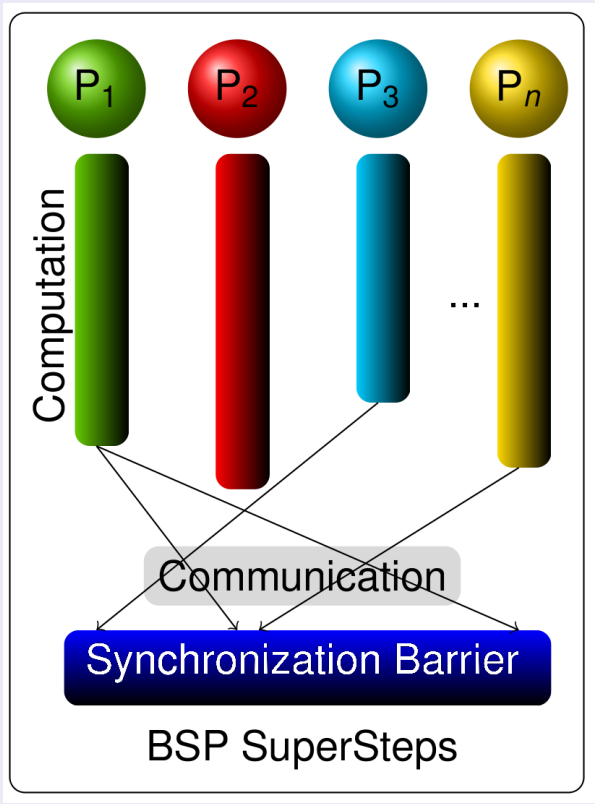
BSML

Un langage fonctionnel

\+ une structure de données parallèle (sans imbrication)

\+ des opérations parallèles sur cette structure

Modèle d'exécution BSP



Thèse

Profiter de la structure d'un langage BSP fonctionnel pour faciliter la preuve de correction de programmes parallèles.

Utiliser les squelettes algorithmiques pour écrire des programmes parallèles corrects par construction.

Outils

Nos moyens

- L'assistant de preuve Coq
- Les primitives parallèles du langage BSML

Vue d'ensemble

Primitives BSML

Squelettes algorithmiques

Programmes et preuves de correction

Extraction

Programmes BSML extraits

Implantation des primitives BSML

Compilation BSML

Exécutions parallèles

Preuve de programme dans un assistant de preuve

Plongement profond

- Arbre de syntaxe abstraite

\+ règles sémantiques

Plongement superficiel

- Langage de l'assistant de preuve

= langage de programmation

\+ axiomatisation

L'assistant de preuve Coq

Programme

```
Fixpoint sum (n :nat) : nat :=
  match n with
    0 ⇒ 0
    | S n' ⇒ n + sum n'
  end.
```

Lemme

```
Lemma sumSpec :
  ∀ n, sum n = n*(1+n)/2.
Proof.
  induction n.
  simpl.
  auto with arith.

  simpl.
  rewrite IHn.
  :
  : omega.
Qed.
```

L'assistant de preuve Coq

Programme fortement spécifié + Russel

```
Program Fixpoint sum' (n :nat) :
  {s : nat | s = n*(n+1)/2} :=
  match n with
    0 ⇒ 0
    | S n' ⇒ n + sum' n'
  end.
```

```
Next Obligation.
  ⋮
omega.
Defined.
```

Programme

```
Fixpoint sum (n :nat) : nat :=
  match n with
    0 ⇒ 0
    | S n' ⇒ n + sum n'
  end.
```

L'assistant de preuve Coq

Programme fortement spécifié + Russel

```
Program Fixpoint sum' (n :nat) :
  {s : nat | s = n*(n+1)/2} :=
  match n with
    0 ⇒ 0
    | S n' ⇒ n + sum' n'
  end.
```

```
Next Obligation.
  ⋮
omega.
Defined.
```

Programme

```
Fixpoint sum (n :nat) : nat :=
  match n with
    0 ⇒ 0
    | S n' ⇒ n + sum n'
  end.
```

Extraction

```
let rec sum n = match n with
| O → O
| S n' → plus n (sum n')
```

Plongement de BSML en Coq

Vecteur parallèle

- Un type abstrait polymorphe : 'a par
- Une taille fixe p : chaque processeur a une valeur de type 'a
- Emboîtement de vecteurs interdite
- Notation informelle : 


```
Module Type PRIMITIVES.

  Parameter bsp_p : nat.

  Axiom bsp_pLtZero : 0 < bsp_p.

  Definition processor : Type := { pid : nat | pid < bsp_p }.

  Parameter par : Type → Type.

⋮
```

Création d'un vecteur parallèle

- **mkpar** : (int → 'a) → 'a par

(**mkpar** f)

$(f\ 0)$	$(f\ 1)$	$\cdots$	$(f\ (p-1))$

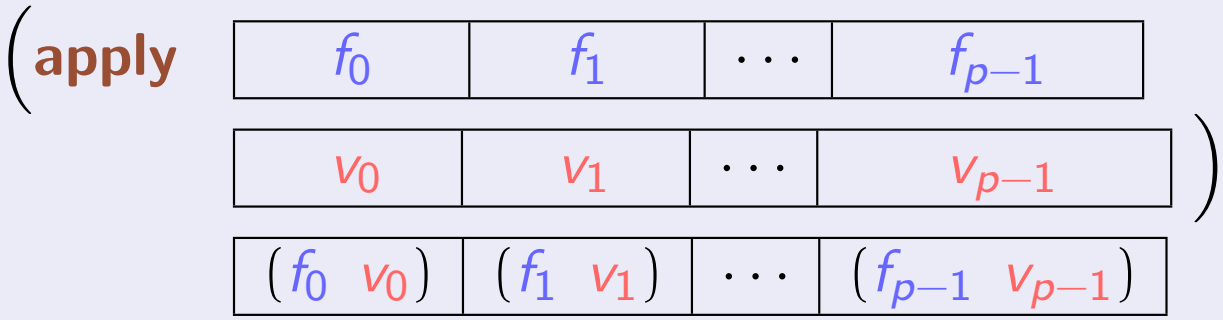
⋮

```
Parameter get : ∀ A : Type, par A → processor → A.

Parameter mkpar : ∀ (A :Type),
  ∀ (f : processor → A),
    { X : par A | ∀ i : processor, get X i = f i }.
```

⋮

Application parallèle point à point

- **apply** : ('a → 'b) par → 'a par → 'b par



⋮

```
Parameter apply : ∀ (A B :Type),
  ∀ (vf : par (A → B)) (vx : par A),
    { X : par B | ∀ i : processor,
      get X i = (get vf i) (get vx i) }.
```

⋮

Communications

- **put**: (int → 'a) par → (int → 'a) par

$$\left(\textbf{put} \quad \boxed{f_0 \mid f_1 \mid \cdots \mid f_{p-1}}\right) = \boxed{g_0 \mid g_1 \mid \cdots \mid g_{p-1}}$$

$g_i\ j = f_j\ i$

0	1	2	3
A	B	C	D
$(f_0\ 1)$	$(f_1\ 1)$	$(f_2\ 1)$	$(f_3\ 1)$
$(f_0\ 2)$	$(f_1\ 2)$	$(f_2\ 2)$	$(f_3\ 2)$
$(f_0\ 3)$	$(f_1\ 3)$	$(f_2\ 3)$	$(f_3\ 3)$

$\Rightarrow$

0	1	2	3
A	$(f_0\ 1)$	$(f_0\ 2)$	$(f_0\ 3)$
B	$(f_1\ 1)$	$(f_1\ 2)$	$(f_1\ 3)$
C	$(f_2\ 1)$	$(f_2\ 2)$	$(f_2\ 3)$
D	$(f_3\ 1)$	$(f_3\ 2)$	$(f_3\ 3)$

⋮

```
Parameter put :
  ∀ (vf : par (processor → A)),
    { X : par (processor → A) |
      ∀ i j : processor, get X i j = get vf j i }.
```

Communications - du local au global

- **proj**: 'a par → (int → 'a)

$$\left(\textbf{proj}\ \boxed{v_0 \mid v_1 \mid \cdots \mid v_{p-1}}\right) = \textbf{function}\ \begin{array}{lcl} 0 & \rightarrow & v_0 \\ 1 & \rightarrow & v_1 \\ \vdots & & \\ p-1 & \rightarrow & v_{p-1} \end{array}$$

⋮

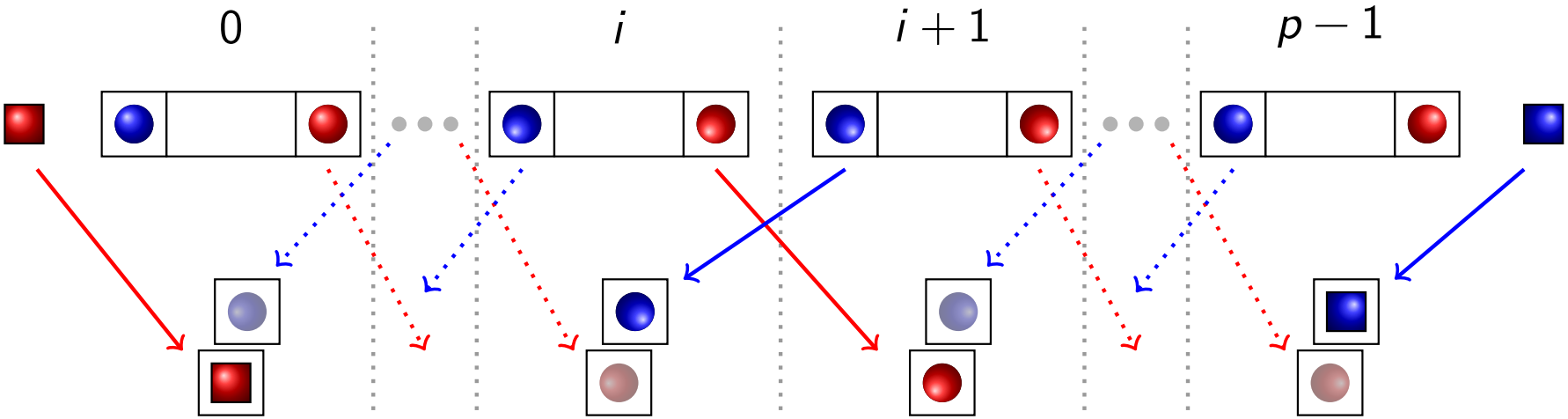
```
Parameter proj :
  ∀ (v : par A),
    { X : processor → A | ∀ i : processor, X i = get v i }.

End PRIMITIVES.
```

Apports

	F. Gava	L. Gesbert	J. Tesson
Version de Coq	7.3	8	8.3
Modélisation	Axiomes	Axiomes	Type de Module
Séparation axiomes/propriétés	X	X	✓
Extraction	X	X nécessite une retouche du code extrait	✓
Réalisation séquentielle	Utilise axiomes		✓

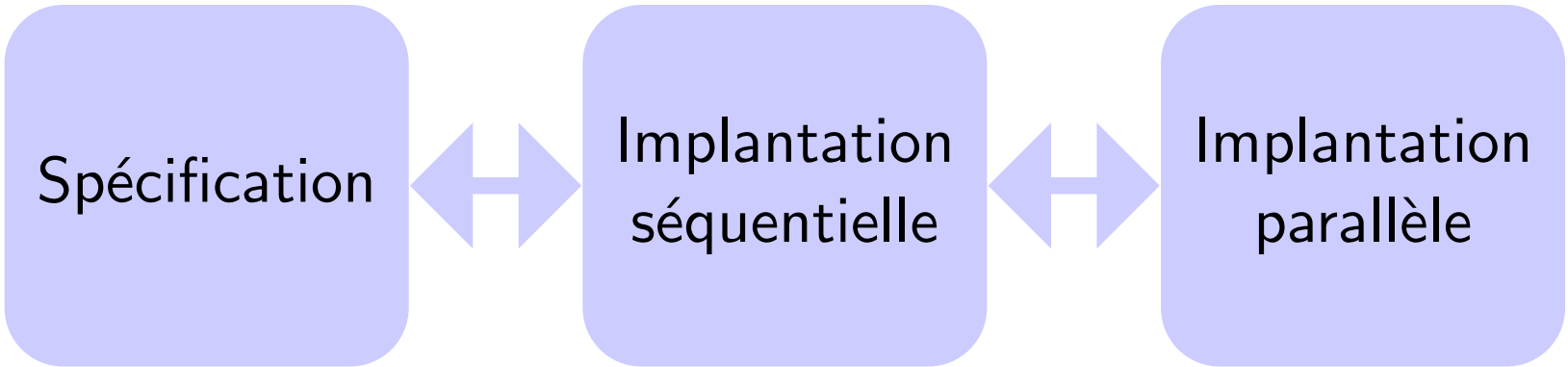
Exemple - Spécification forte d'une fonction

```
Program Definition getBounds (A :Type)(l r : A)
  (v : par(list A))(H : ∀ i, get v i ≠ nil) :
  { vl : par A | ∀ (i :processor), get vl i =
      if ( i == firstProc ) then l else sLast (get v (i-1))} ×
  { vr : par A | ∀ (i :processor), get vr i =
    if ( i == lastProc) then r else
                    sHead (get v (min (i+1) lastProc)) }
```



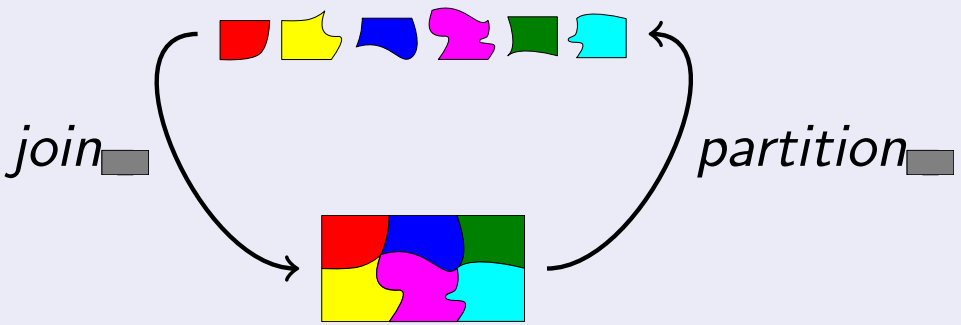
Exemple - Spécification forte d'une fonction

```
Program Definition getBounds (A :Type)(l r : A)
  (v : par(list A))(H : ∀ i, get v i ≠ nil) :
  { vl : par A | ∀ (i :processor), get vl i =
      if ( i == firstProc ) then l else sLast (get v (i-1))} ×
  { vr : par A | ∀ (i :processor), get vr i =
    if ( i == lastProc) then r else
                    sHead (get v (min (i+1) lastProc)) }
```



Répartition des données

Structure de données partitionnable

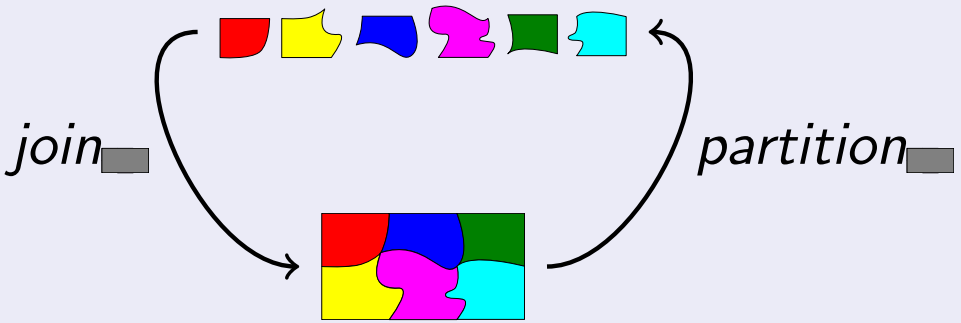


Propriétés

$partition_{\blacksquare} \circ join_{\blacksquare} =$

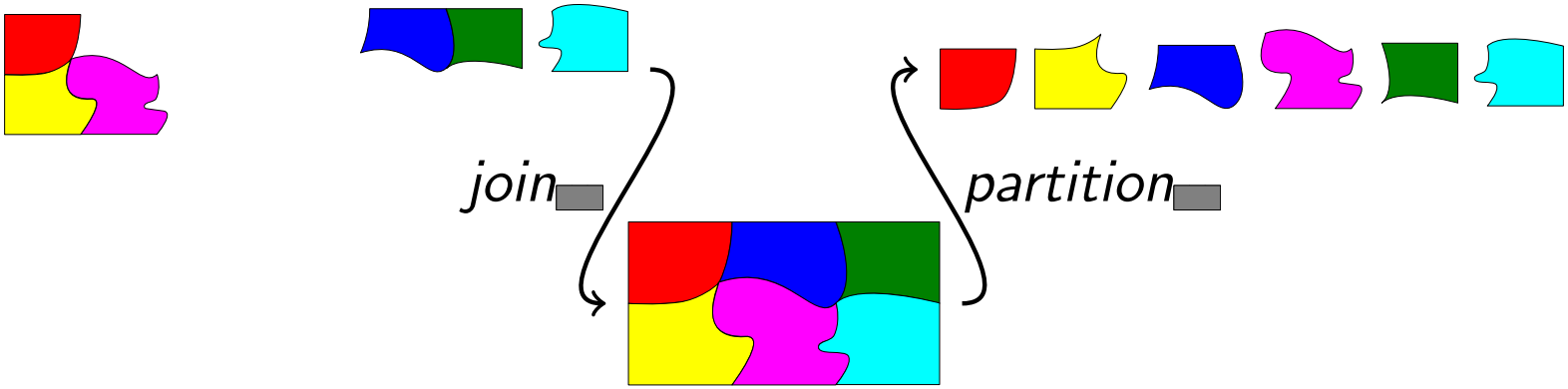
Répartition des données

Structure de données partitionnable



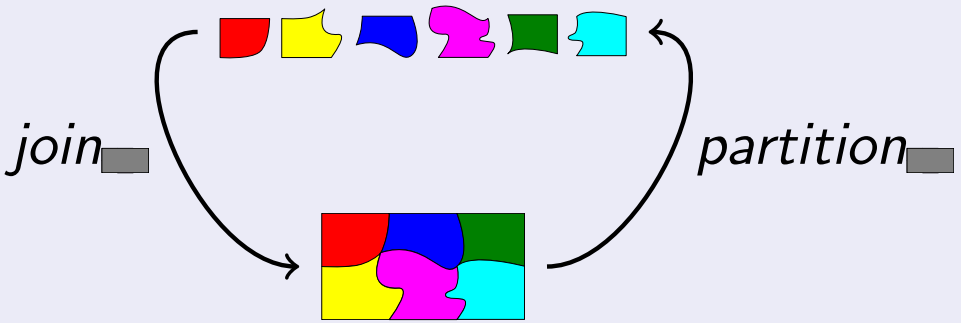
Propriétés

$partition_{\blacksquare} \circ join_{\blacksquare} = ?$



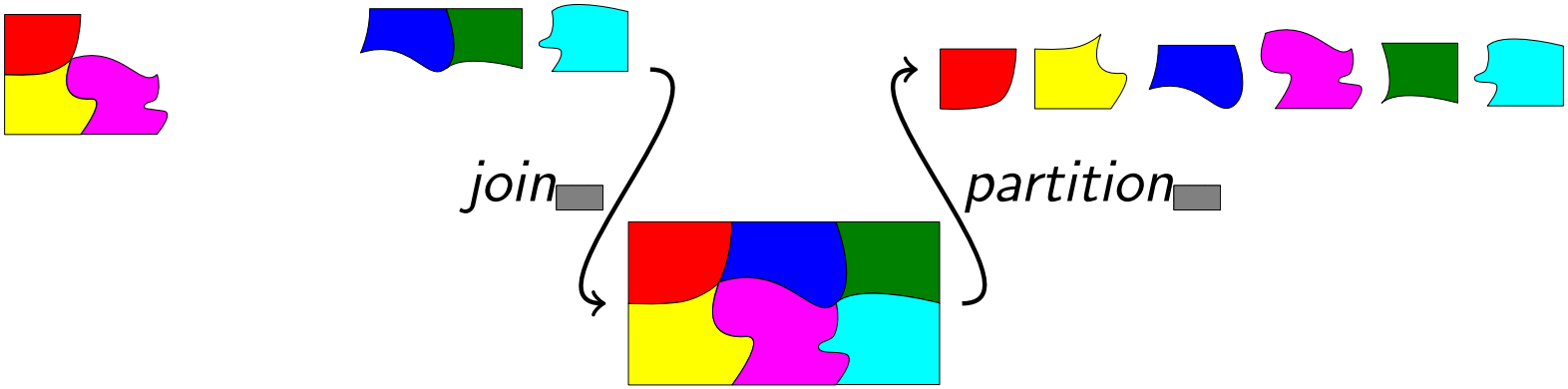
Répartition des données

Structure de données partitionnable



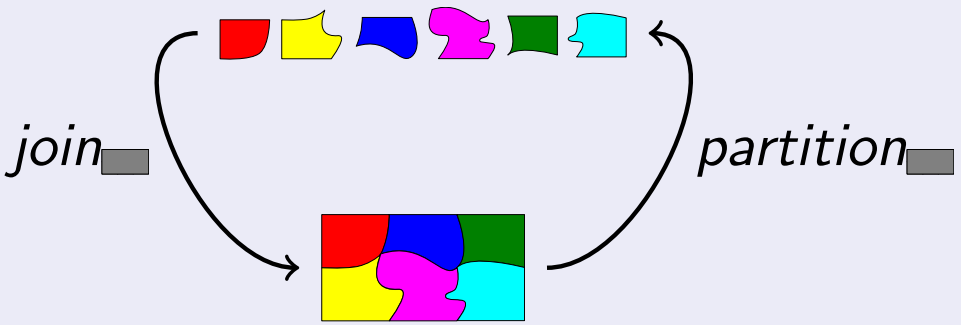
Propriétés

$partition_{\blacksquare} \circ join_{\blacksquare} =$



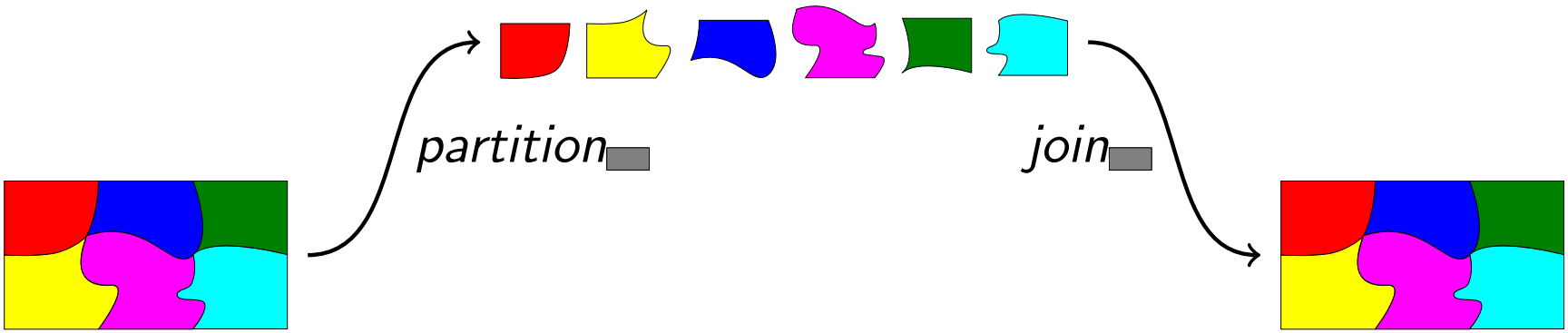
Répartition des données

Structure de données partitionnable



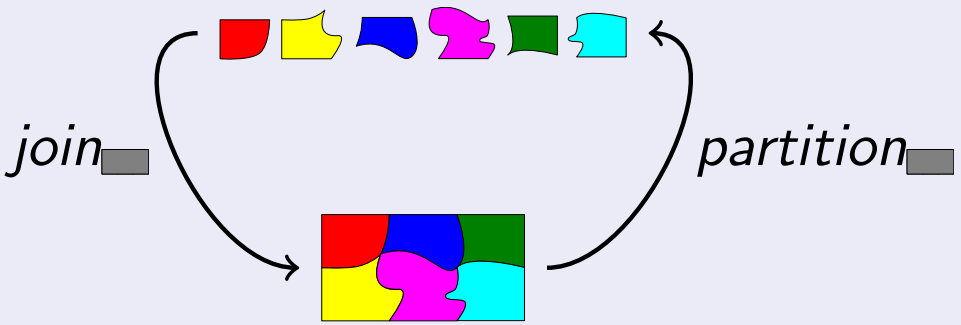
Propriétés

$join_{\blacksquare} \circ partition_{\blacksquare} = ?$



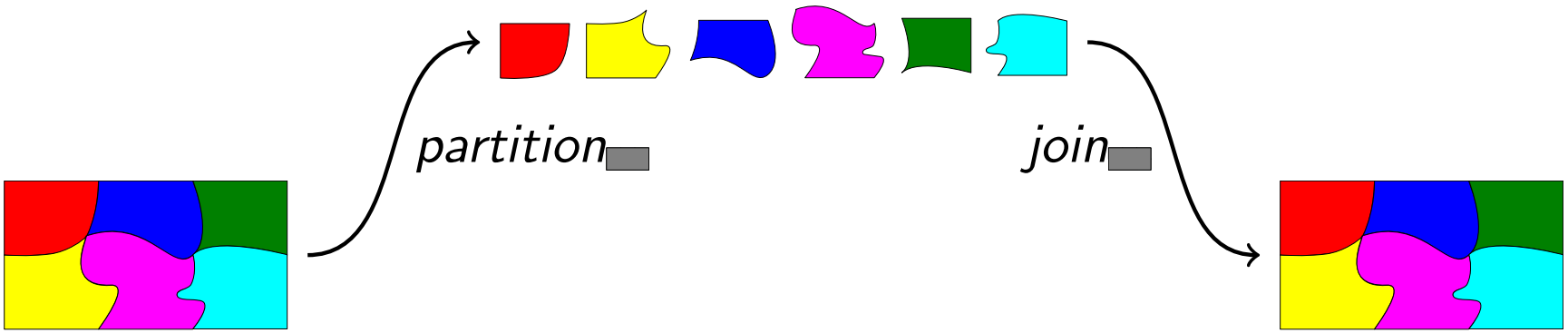
Répartition des données

Structure de données partitionnable

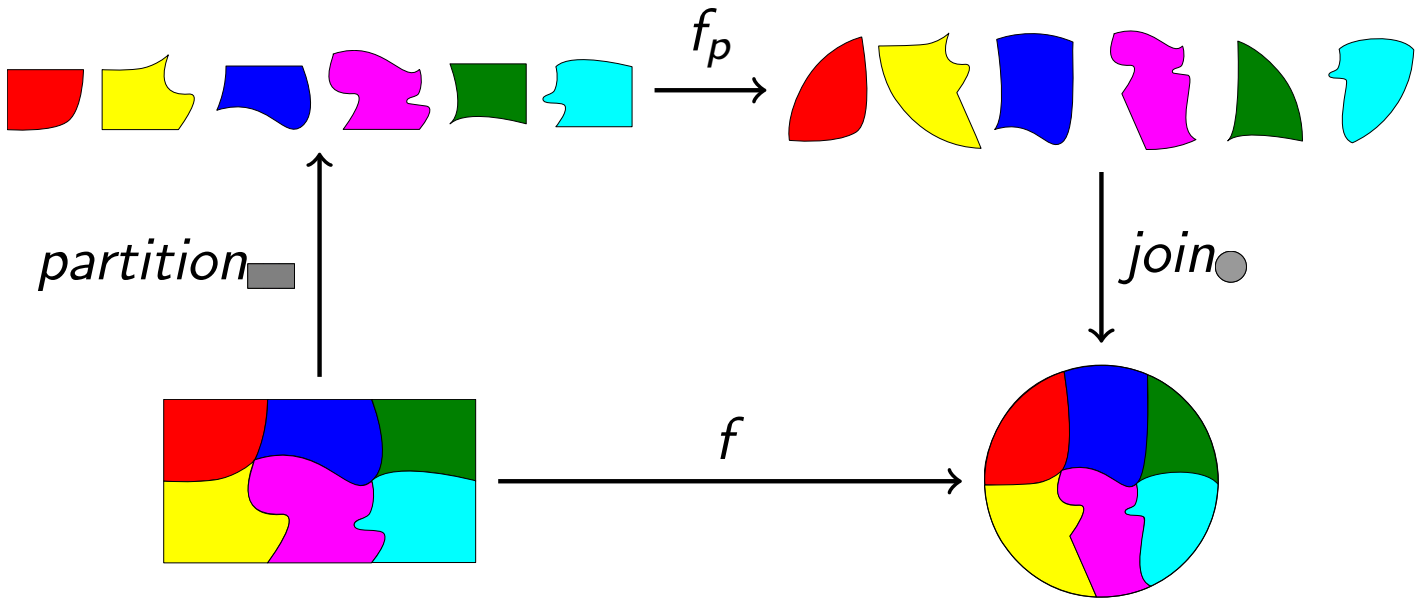


Propriétés

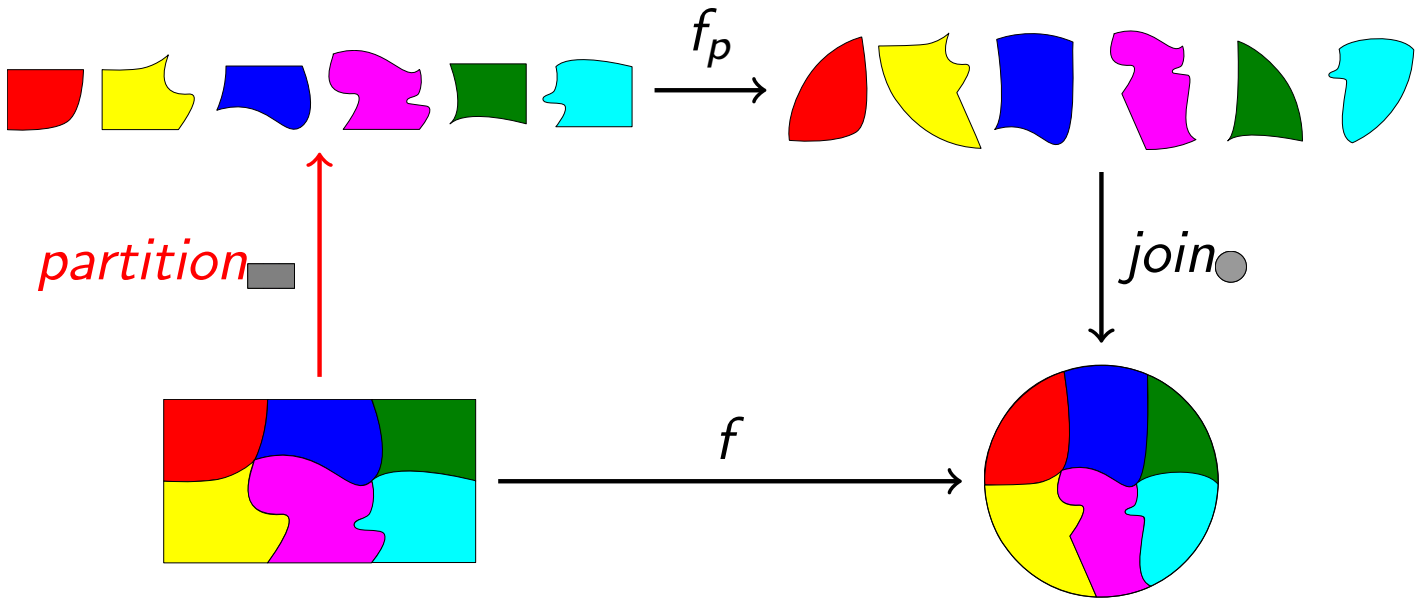
$join_{\blacksquare} \circ partition_{\blacksquare} = id_{\blacksquare}$



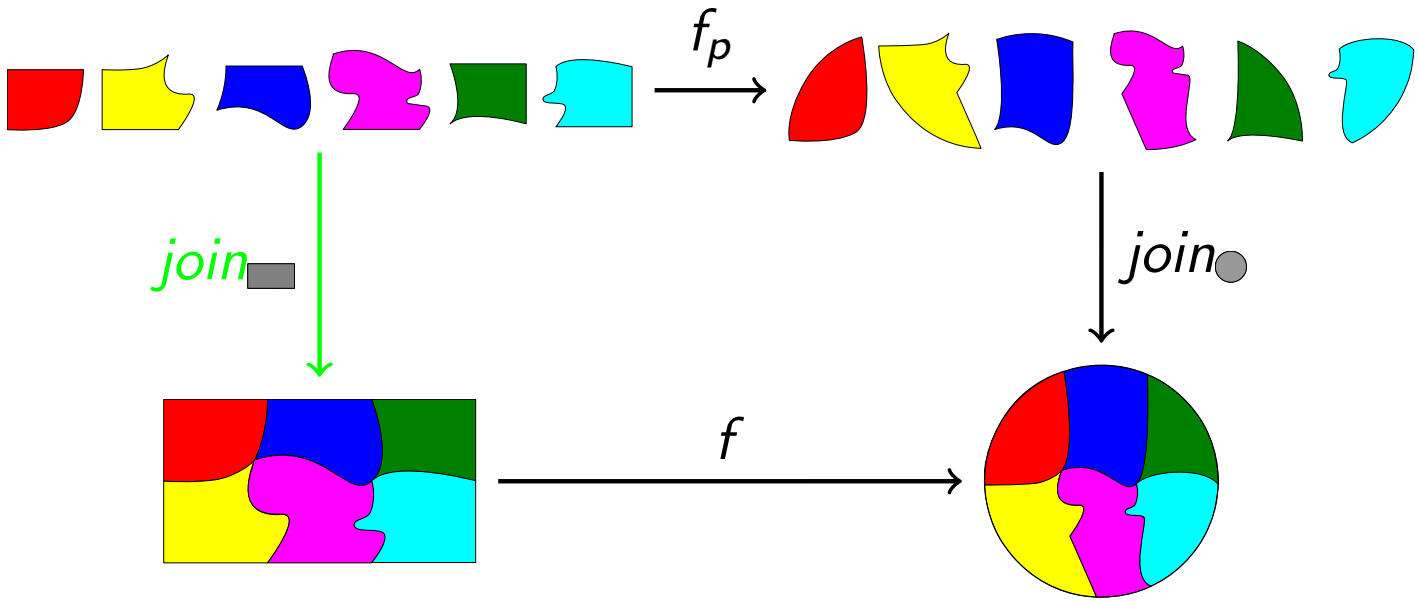
Parallélisation correcte



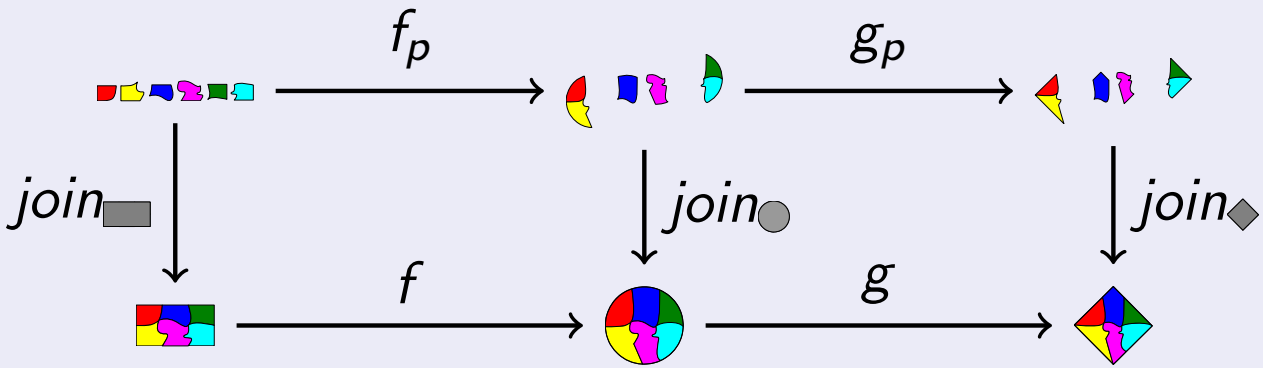
Parallélisation correcte



Parallélisation correcte



Parallélisation correcte composable



Classes de type en Coq - surcharge

```
Class Associative (op : t→t→t) :=
{    associative : ∀ e1 e2 e3 : t,
        op (op e1 e2) e3 =
        op e1 (op e2 e3)
}.
```

```
Instance plus_associatif : Associative
nat plus :=
{associative := plus_assoc}.

Instance mult_associatif : Associative
nat mult.
constructor.
apply mult_assoc.
Qed.
```

```
Print Instances Associative.
```

```
  mult_associatif :
Associative nat mult.

  plus_associatif :
Associative nat plus.
```

Classes de type en Coq - surcharge

```
Class Associative (op : t→t→t) :=
{   associative : ∀ e1 e2 e3 : t,
      op (op e1 e2) e3 =
      op e1 (op e2 e3)
}.
```

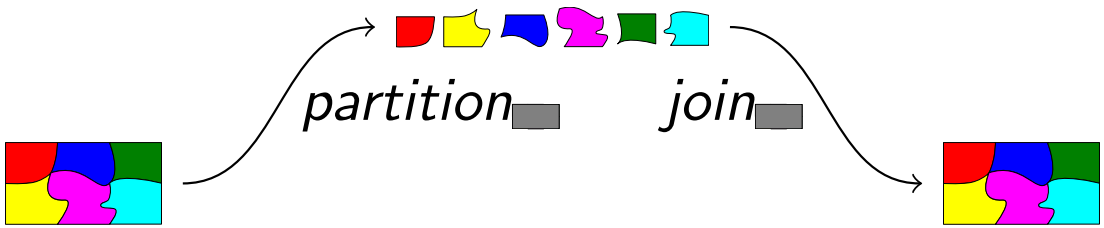
```
Print Instances Associative.
  mult_associatif :
Associative nat mult.

  plus_associatif :
Associative nat plus.
```

```
Goal ∀ a b c : nat, a×b×c+b+c = a×(b×c)+(b+c).
  intros.
  replace (a×b×c) with (a×(b×c)) by (apply associative).
  replace (a×(b×c)+b+c) with (a×(b×c)+(b+c))
                                  by (apply associative).
  reflexivity.
Qed.
```

Répartition - Formalisation Coq I

Répartition d'un type

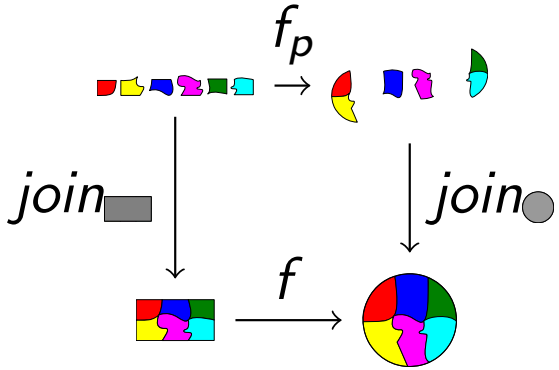
```
Class Partitionnable (A : Type) :=
{
  parallel_type : Type;
  join : parallel_type → A;
  partition : A → parallel_type;
  join_part_match : ( ∀ a : A, join ( partition a ) = a )
}.
```



Parallélisation correcte composable - Coq I

Parallélisation correcte

```
Class Parallel `{A_Part : Partitionnable A}
  `{B_Part : Partitionnable B}
  (f : A → B)
  (fp : parallel_type(A :=A) → parallel_type(A :=B)) :=
{
  parallel_spec_match : ∀ par_a,
        join ( fp (par_a)) = f (join par_a)
}.
```



Parallélisation correcte composable - Coq II

Instance

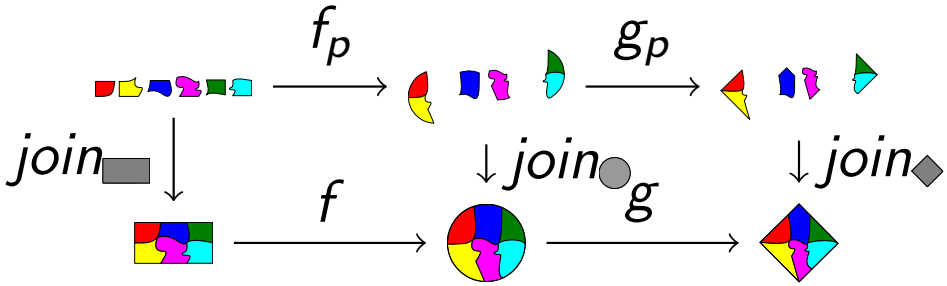
```
Instance FilterParCorrect
(E :Type)(select :E→bool) :Parallel(filter select)(filterPar select).
Proof.
...
```

Pour déclarer une implantation parallèle, il faut vérifier la condition *parallel_spec_match*.

Parallélisation correcte composable - Coq III

Composition correcte

```
Instance correctComposition (A B C : Type)
  (APart : Partitionnable A)
  (BPart : Partitionnable B)
  (CPart : Partitionnable C)
  (f : B → C) fp (parf : Parallel f fp)
  (g : A → B) gp (parg : Parallel g gp) :
    Parallel (f :o: g) (fp :o: gp).
```



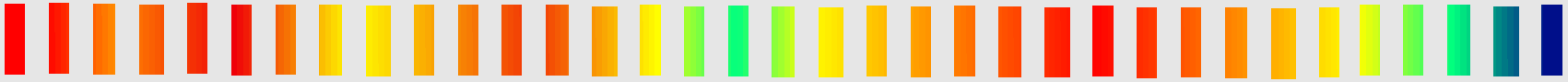
Diffusion de la chaleur 1D

Équation de diffusion :

$$\frac{\delta u}{\delta t} - \kappa \frac{\delta^2 u}{\delta^2 x} = 0 \qquad \forall t,\ u(0,t) = l \qquad \forall t,\ u(1,t) = r$$

- κ est le coefficient de diffusion du métal,
- l et r des constantes (température aux bords, en dehors du métal)

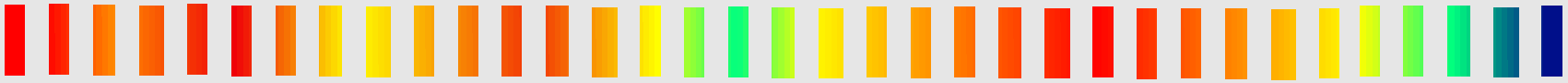
Diffusion de la chaleur 1D



Une version discrétisée :

$$
\begin{aligned}
& u(x, t + dt) \\
& \quad = \\
& \qquad \frac{\kappa dt}{dx^2} \times \big(\\
& \qquad\qquad u(x + dx, t) \\
& \qquad\quad + \quad u(x - dx, t) \\
& \qquad\quad - \quad 2 \times u(x, t) \big) \\
& \qquad + u(x, t)
\end{aligned}
$$

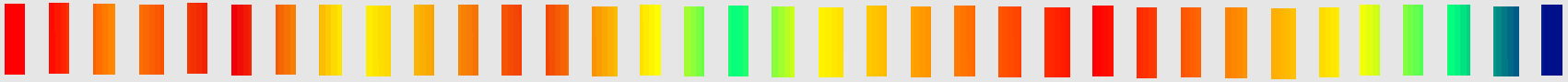
Diffusion de la chaleur 1D



Une version discrétisée :

$$
\begin{aligned}
&(\mathbf{step}\ \kappa\ dx\ dt\ l\ r\ u)\ (x) \\
&\quad = \\
&\qquad \frac{\kappa dt}{dx^2} \times \big(\\
&\qquad\qquad u(x+dx) \\
&\qquad\quad + \quad u(x-dx) \\
&\qquad\quad - \quad 2 \times u(x)\big) \\
&\qquad + u(x)
\end{aligned}
$$

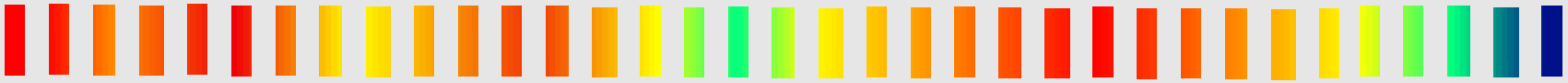
Diffusion de la chaleur 1D



Spécification :

$$
\begin{aligned}
&(\mathbf{step}\ \kappa\ dx\ dt\ l\ r\ u)\ [i] \\
&\quad = \\
&\qquad \frac{\kappa dt}{dx^2} \times \big(\\
&\qquad\qquad u[i+1] \\
&\qquad\quad + \quad u[i-1] \\
&\qquad\quad - \quad 2 \times u[i] \big) \\
&\qquad + u[i]
\end{aligned}
$$

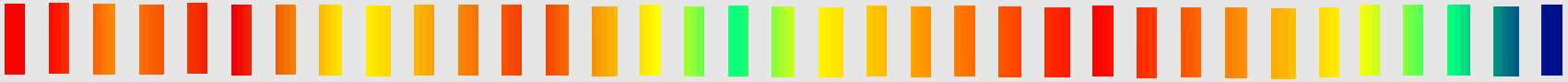
Diffusion de la chaleur 1D



Spécification :

$$
\begin{aligned}
&(\mathbf{step}\ \kappa\ dx\ dt\ l\ r\ u)\ [i] \\
&\quad = \\
&\qquad \tfrac{\kappa dt}{dx^2} \times \big(\\
&\qquad\qquad \mathbf{if}\ i \leq nb\_elem\ \mathbf{then}\ r\ \mathbf{else}\ u[i+1] \\
&\qquad\quad + \quad \mathbf{if}\ i = 0\ \mathbf{then}\ l\ \mathbf{else}\ u[i-1] \\
&\qquad\quad - \quad 2 \times u[i] \big) \\
&\qquad + u[i]
\end{aligned}
$$

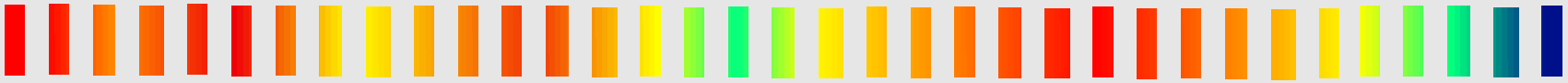
Diffusion de la chaleur 1D



Spécification :

$$
\begin{aligned}
&\mathit{nth}\ i\ (\mathbf{step}\ \kappa\ dx\ dt\ l\ r\ u)\ d \\
&\quad = \\
&\qquad \tfrac{\kappa dt}{dx^2} \times \Big(\\
&\qquad\qquad \mathbf{if}\ i \leq \mathit{nb\_elem}\ \mathbf{then}\ r\ \mathbf{else}\ \mathit{nth}\ (i+1)\ u\ d \\
&\qquad\quad + \quad \mathbf{if}\ i = 0\ \mathbf{then}\ l\ \mathbf{else}\ \mathit{nth}\ (i-1)\ u\ d \\
&\qquad\quad - \quad 2 \times (\mathit{nth}\ i\ u\ d)\Big) \\
&\qquad + (\mathit{nth}\ i\ u\ d)
\end{aligned}
$$

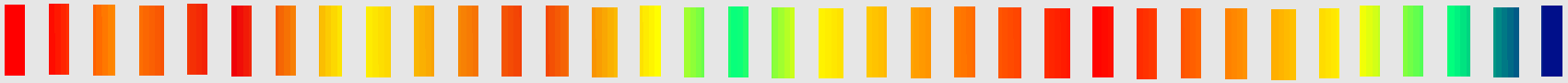
Diffusion de la chaleur 1D



Spécification :

$$
\begin{array}{l}
nth\ i\ (\mathbf{step}\ \kappa\ dx\ dt\ l\ r\ u)\ d \\
\quad = \\
\qquad \frac{\kappa dt}{dx^2} \times \big(\\
\qquad\qquad nth\ (i+1)\ u\ r \\
\qquad\quad + \quad \mathbf{if}\ i = 0\ \mathbf{then}\ l\ \mathbf{else}\ nth\ (i-1)\ u\ d \\
\qquad\quad - \quad 2 \times (nth\ i\ u\ d)\big) \\
\qquad + (nth\ i\ u\ d)
\end{array}
$$

Diffusion de la chaleur 1D



Spécification :

```
Definition StepSpecification step : Prop :=
    ∀ (u : list number)(Hu : u≠[])(l r dt dx κ : number)
      (i : nat)(Hi :i<length u)(d : number),
nth i (step κ dt dx l r u) d
    =
        κ × dt / (dx × dx)(
                nth (i+1) u r
          +     if beq_nat i 0 then l else nth (i−1) u d
          −   (nth i u d) + (nth i u d)
        + (nth i u d)
```

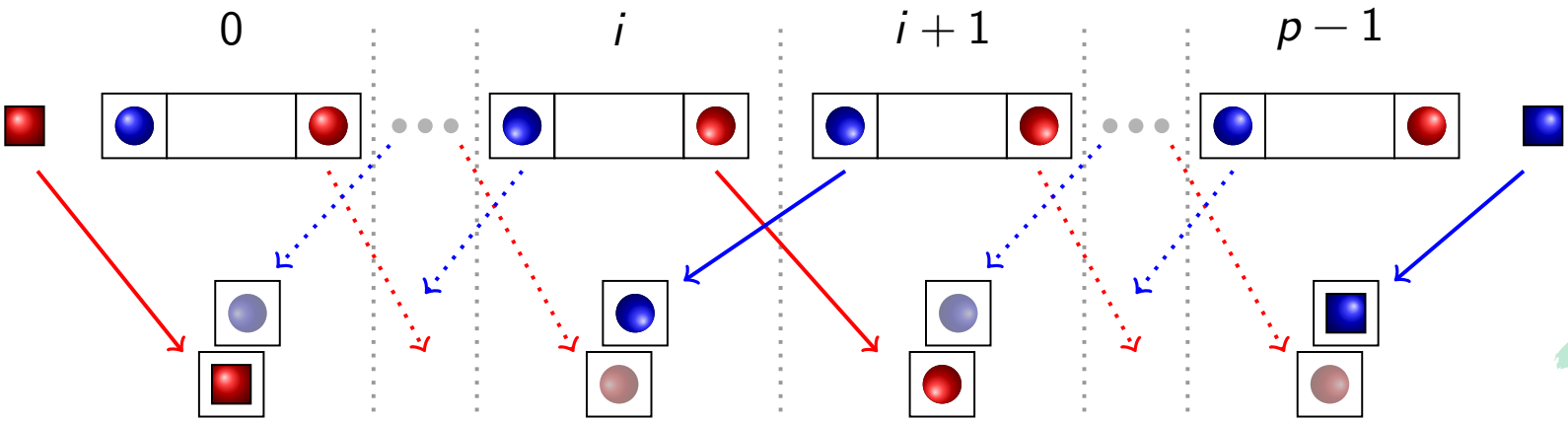
Diffusion de la chaleur 1D

Implantation récursive purement fonctionnelle

```
Fixpoint heatSeq l r dt dx κ (u : list number) : list number :=
    := ...
```

Implantation BSML

```
Program Definition heatPar l r dt dx κ(u : par(list number))
      (Hu : ∀ i, get u i ≠ nil)
  : par(list number) :=
  let bounds := getBounds l r Hu in
    apply (parfun2
      (fun l r ⇒ heatSeq l r dt dx κ) (fst bounds) (snd bounds))
    u.
```



Diffusion de la chaleur : correction

Correction de l'implantation séquentielle

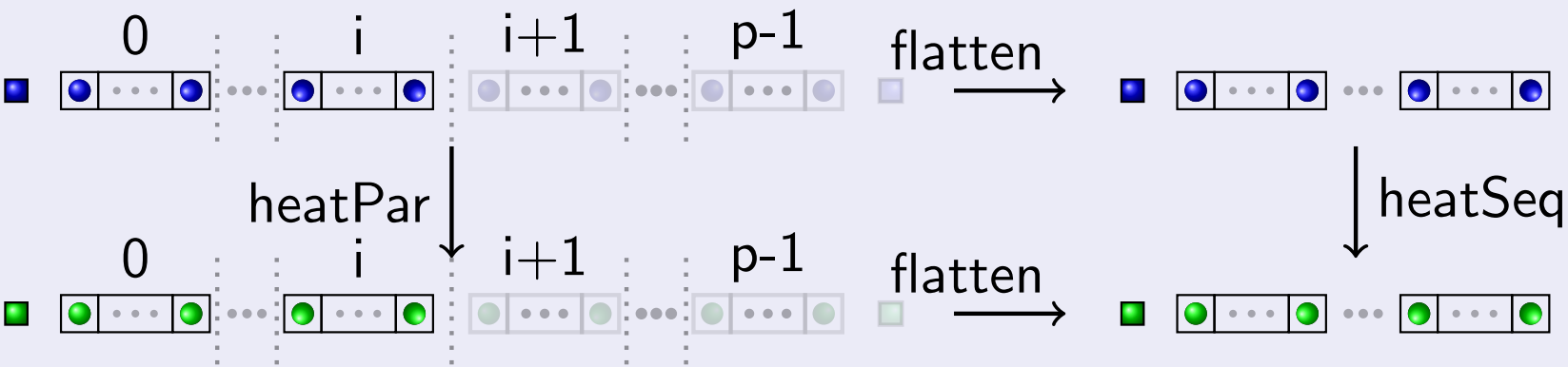
```
Lemma stepSeqFollowsHeatEquationStepSpecification :
      StepSpecification heatSeq.
Proof.
⋮


  induction u
    ⋮
    induction i
```

Diffusion de la chaleur : correction - Version parallèle

Condition de parallélisation

- Version parallèle : au moins un élément par processeur
- Version séquentielle : liste de taille supérieure au nombre de processeurs

Lemme de séquentialisation

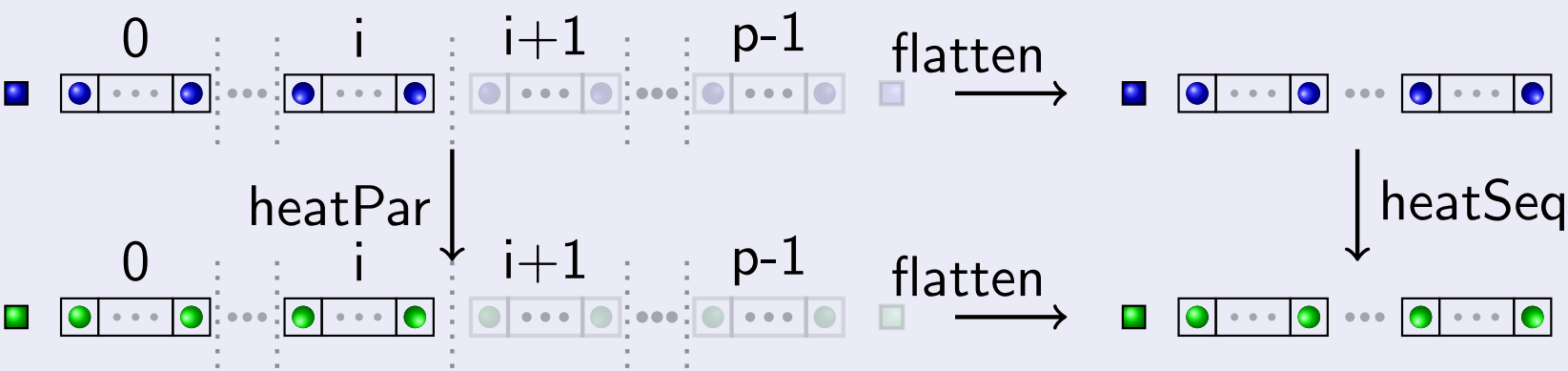


Théorème

La version parallèle est une parallélisation correcte composable de l'implantation séquentielle : résultat précédant avec $i = p - 1$

Diffusion de la chaleur : correction - Version parallèle

Lemme de séquentialisation



Théorème

La version parallèle est une parallélisation correcte composable de l'implantation séquentielle : résultat précédant avec $i = p - 1$

Théorème - Coq

Parallel *heatSeq heatPar*

Parallélisation automatique

Comptage d'éléments de taille supérieure à 0
counting String *length* ["abc" ;"" ;"" ;"a" ;"ab"] = 3

```
Definition counting (A :Type) (size : A→ nat) :=
(fun l ⇒ fold_left plus l 0):o:(map (fun x ⇒ 1)):o:
                        (filter (fun x ⇒ x > 0)):o:(map size).

Definition countingPar (A :Type) (size : A→ nat) :=
        parallel (f :=counting A size).
```

Parallélisation automatique

$$parallel\ (f\ :=counting\ A\ size)$$

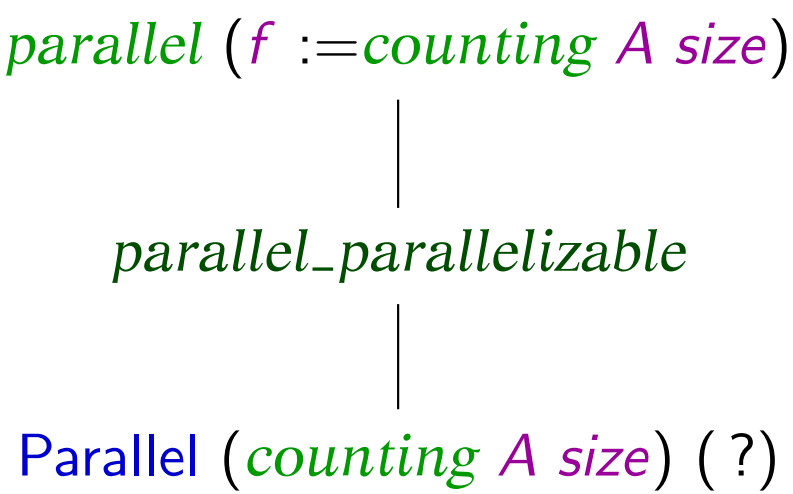
Parallélisation automatique

$$parallel\ (f := counting\ A\ size)$$

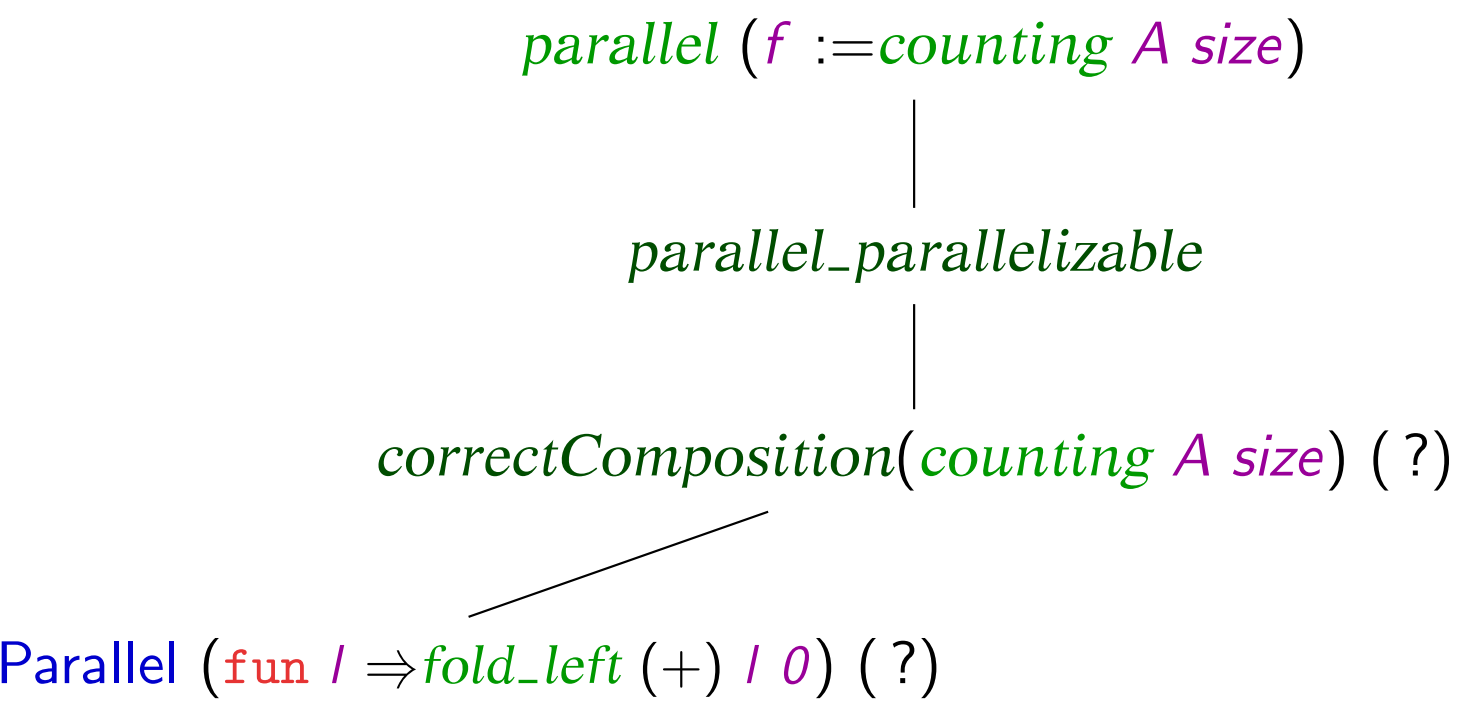
$$\Big|$$

$$\text{Parallelizable}\ (counting\ A\ size)$$

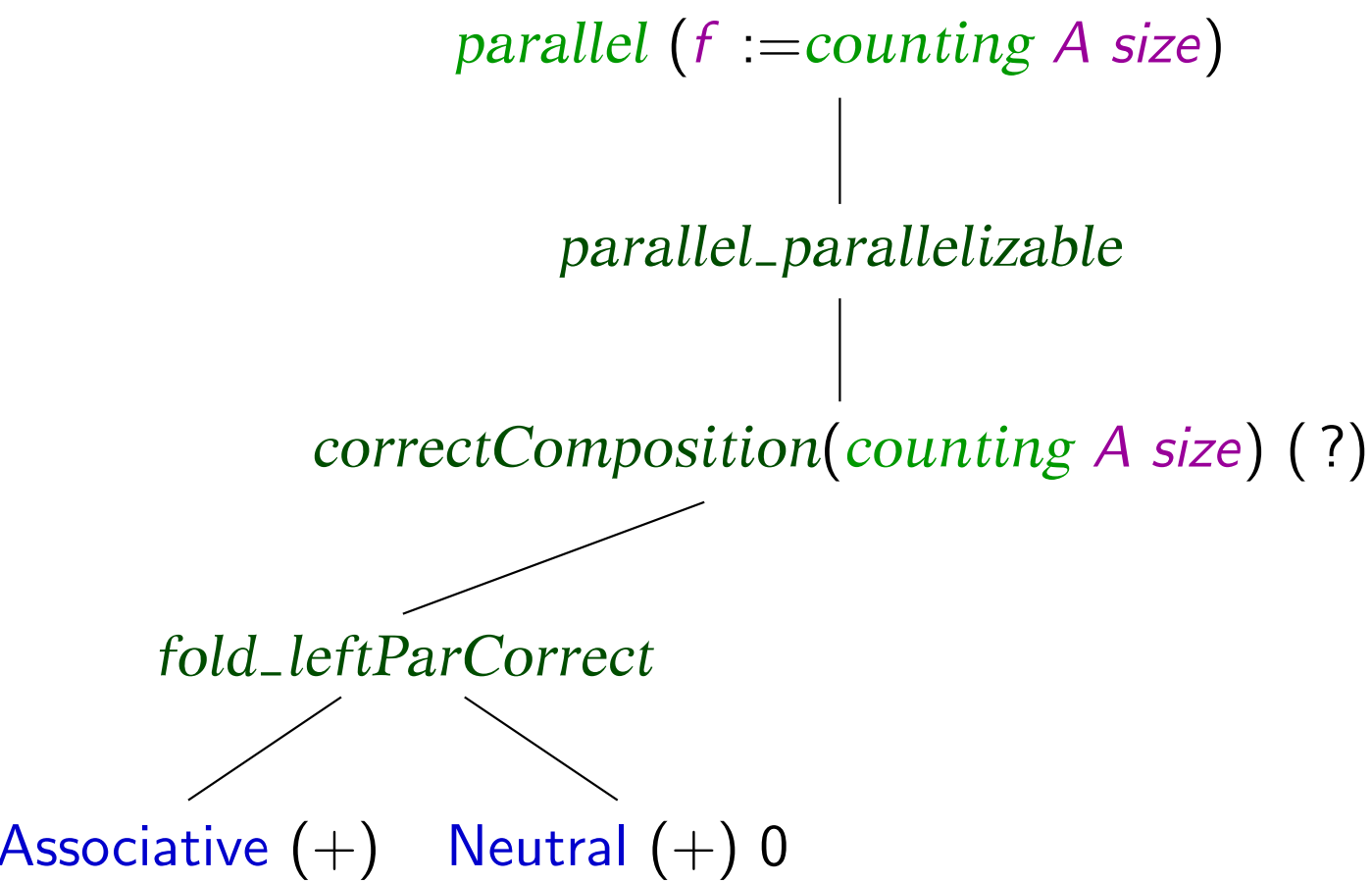
Parallélisation automatique



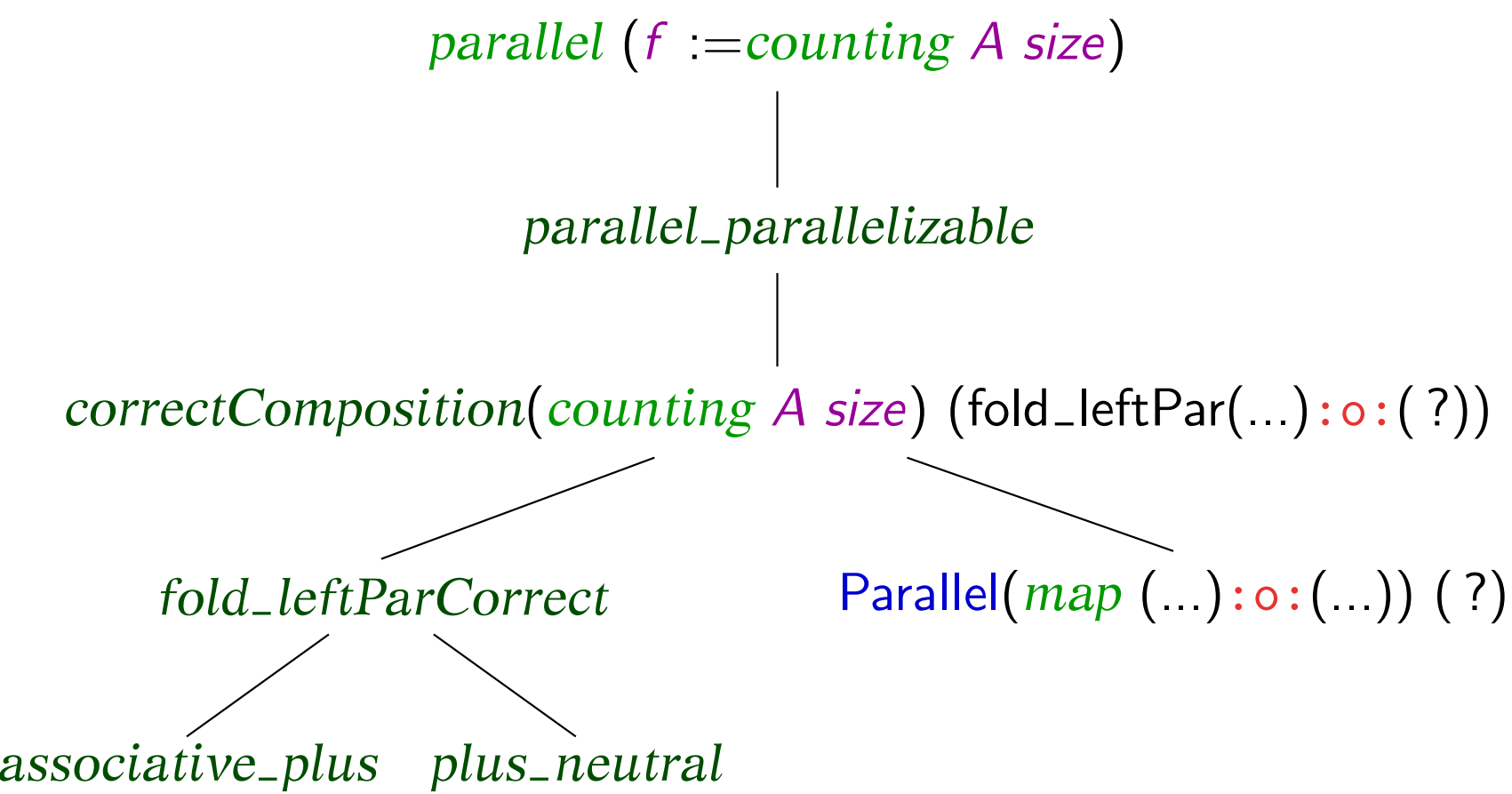
Parallélisation automatique



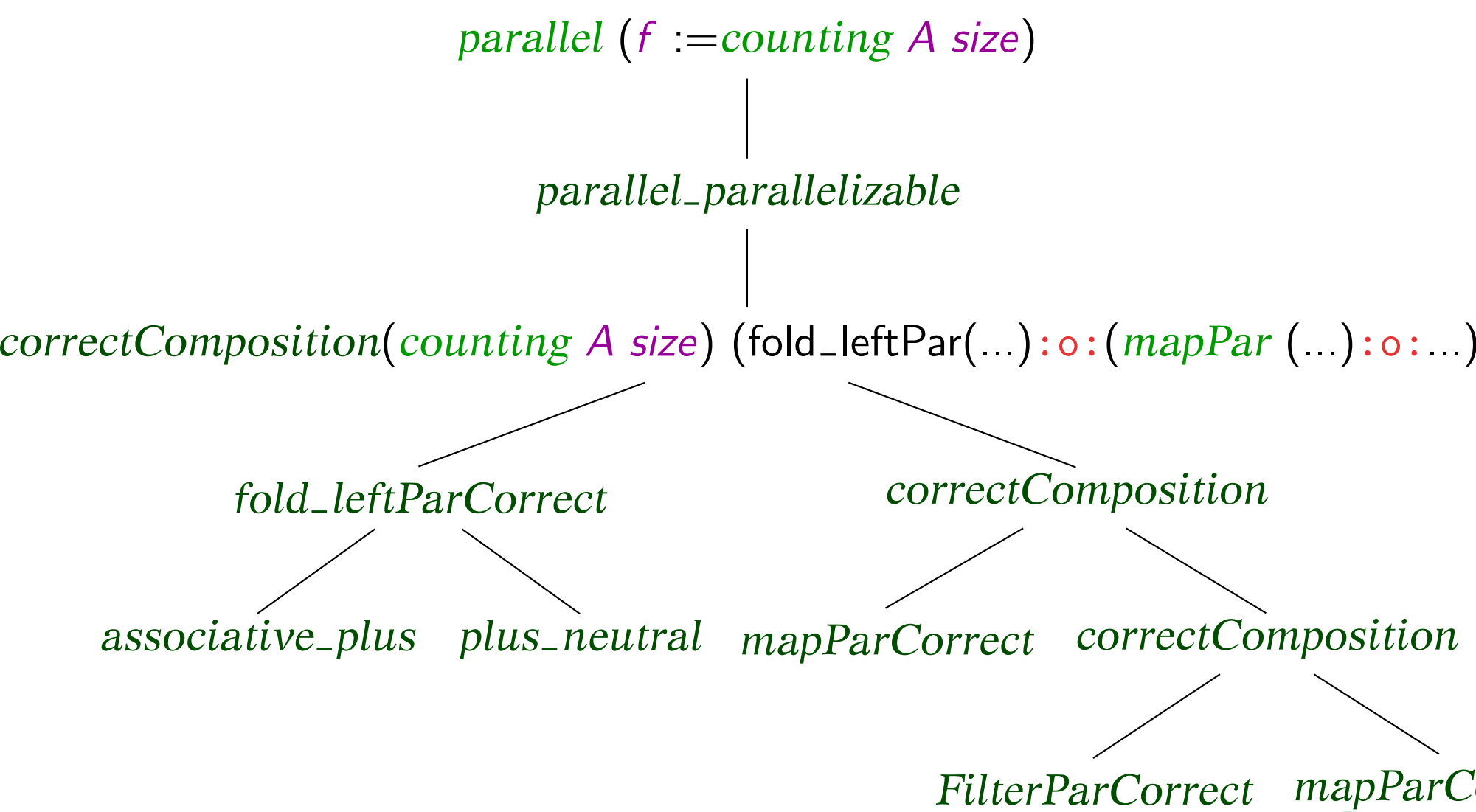
Parallélisation automatique



Parallélisation automatique

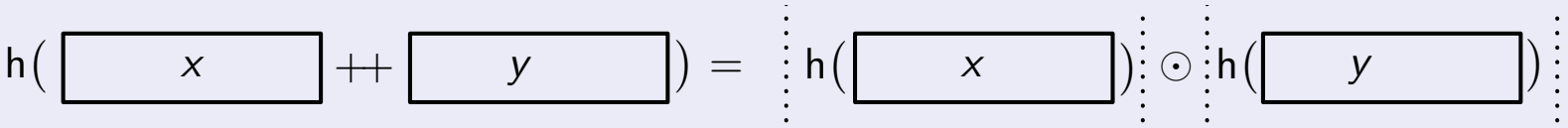


Parallélisation automatique

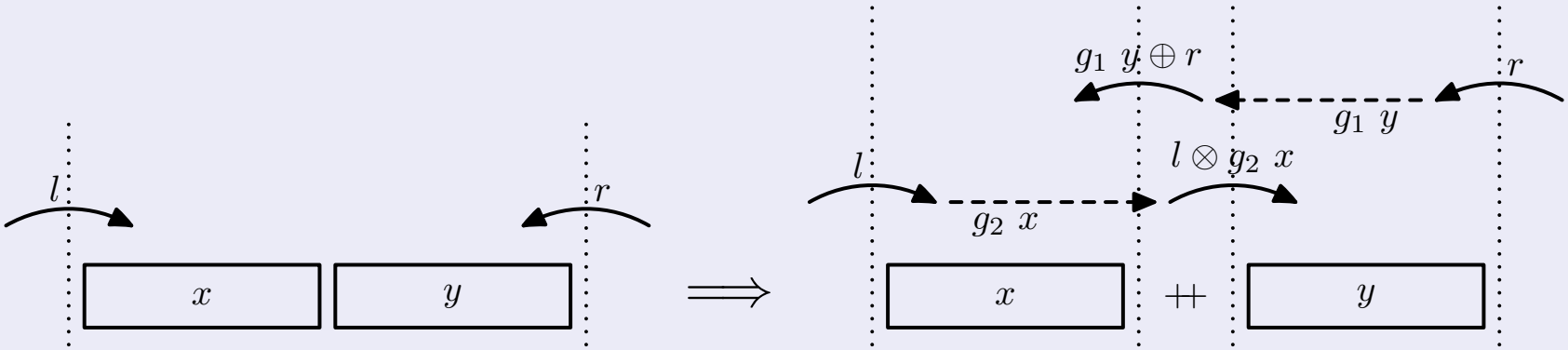


Homomorphismes de listes pour le parallélisme

Homomorphisme de listes



Homomorphisme BSP - Hu & Gesbert 2008



L'homomorphisme BSP : formellement

Définition (BH)

La fonction h est un homomorphisme BSP, ou BH, si elle vérifie les équations suivantes :

$$\begin{cases} bh\ []\ l\ r = [] & (BH\text{-nil}) \\ bh\ [a]\ l\ r = [k\ a\ l\ r] & (BH\text{-Singleton}) \\ bh\ (x +\!\!+ y)\ l\ r = bh\ x\ l\ (g_r\ y \otimes_r\ r)\ +\!\!+ & \\ \qquad bh\ y\ (l\ \oplus_l\ g_l\ x)\ r & (BH\text{-Concat}) \end{cases}$$

Conditions sur g_l, g_r, $\oplus_l$ et $\otimes_r$

$(g_l\ [])$ neutre pour $\oplus_l$ & $(g_r\ [])$ neutre pour $\otimes_r$

$$(g_r\ (y +\!\!+ z)) \otimes_r\ r = (g_r\ y) \otimes_r ((g_r\ z) \otimes_r\ r) \quad (g_r\text{-Concat-faible})$$

$$l \oplus_l (g_l\ (x +\!\!+ y)) = (l \oplus_l (g_l\ x)) \oplus_l (g_l\ y) \quad (g_l\text{-Concat-faible})$$

L'homomorphisme BSP : formellement

Définition (BH)

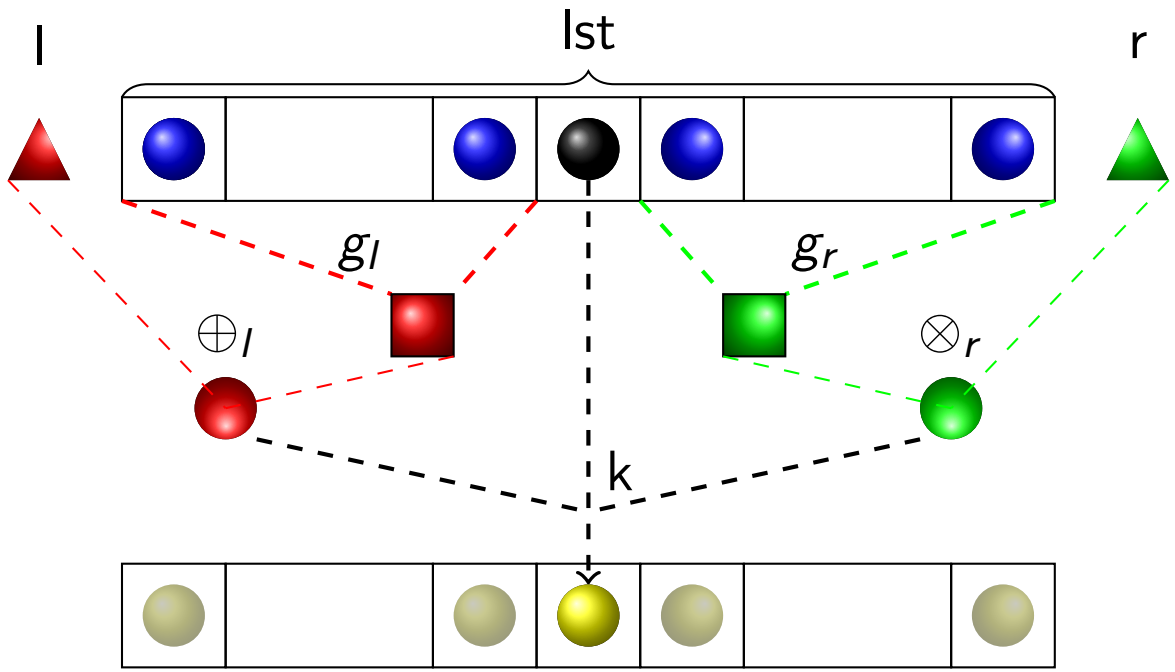
La fonction h est un homomorphisme BSP, ou BH, si elle vérifie les équations suivantes :

$$\begin{cases} bh\ []\ l\ r = [] & (BH\text{-nil}) \\ bh\ [a]\ l\ r = [k\ a\ l\ r] & (BH\text{-Singleton}) \\ bh\ (x +\!\!+ y)\ l\ r = bh\ x\ l\ (g_r\ y \otimes_r\ r)\ +\!\!+ & \\ \qquad bh\ y\ (l\ \oplus_l\ g_l\ x)\ r & (BH\text{-Concat}) \end{cases}$$

Conditions suffisantes sur g_l, g_r, $\oplus_l$ et $\otimes_r$

g_l, g_r sont des homomorphismes dont les opérateurs associés sont $\oplus_l$ et $\otimes_r$

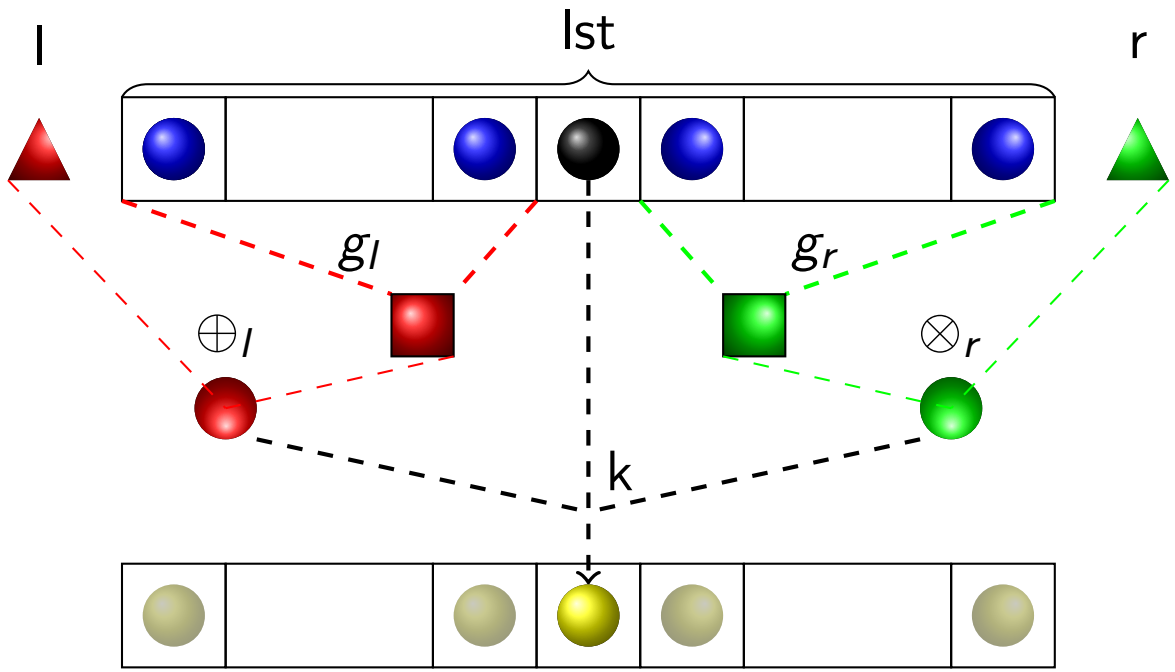
L'homomorphisme BSP - Calcul séquentiel



$$\forall\ \text{bh}\ l\ lst\ r\ i,\ nth\ i\ (\text{bh}\ l\ lst\ r) =$$
$$k$$
$$(l \oplus_l (gl\ (nth\ i\ (inits\ lst))))$$
$$(nth\ i\ lst)$$
$$(gr\ (nth\ i\ (tails\ lst))) \otimes_r r)$$

L'homomorphisme BSP - Calcul séquentiel

- *bh_comp*



$$\forall\ \text{bh}\ l\ lst\ r\ i,\ nth\ i\ (\text{bh}\ l\ lst\ r) =$$
$$k$$
$$(l \oplus_l (gl\ (nth\ i\ (inits\ lst))))$$
$$(nth\ i\ lst)$$
$$(gr\ (nth\ i\ (tails\ lst))) \otimes_r r)$$

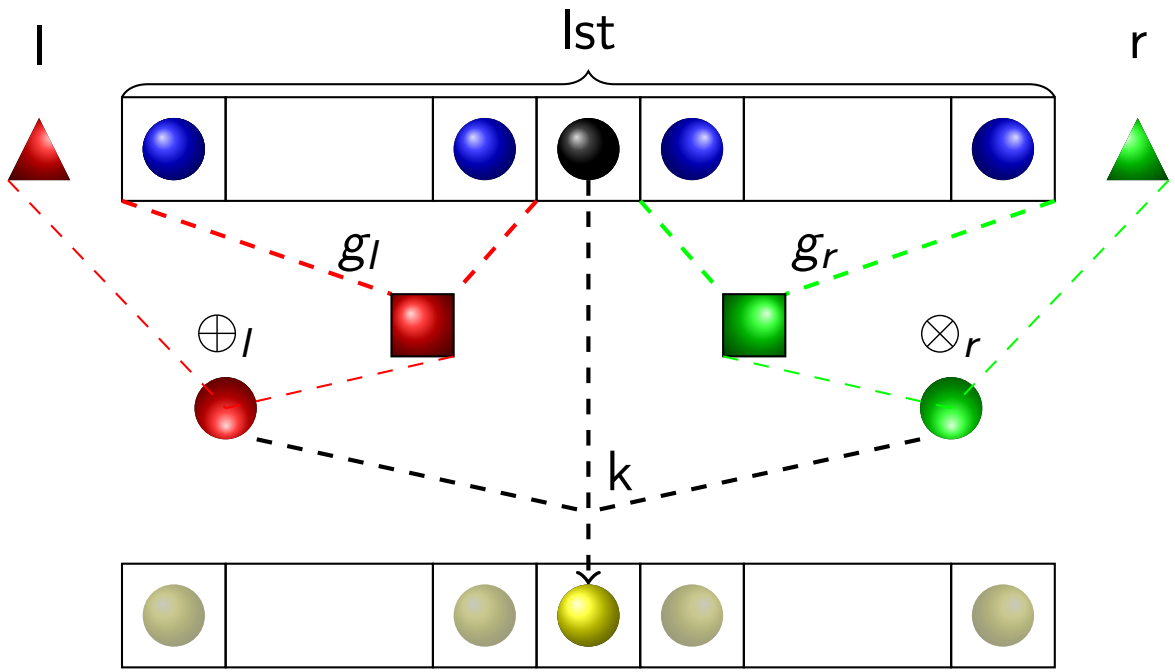
L'homomorphisme BSP - Calcul séquentiel

- *bh_comp*

  ==

- *bh_seq*

map3 k
  (*map* g_l (*inits* l))
  l
  (*map* g_r (*tails* l))



$\forall$ bh *l lst r i*, *nth i* (bh *l lst r*) =
k
(*l* $\oplus_l$(*gl* (*nth i* (*inits lst*))))
(*nth i lst*)
(*gr* (*nth i* (*tails lst*))) $\otimes_r$ *r*)

L'homomorphisme BSP - Calcul séquentiel

- *bh_comp*

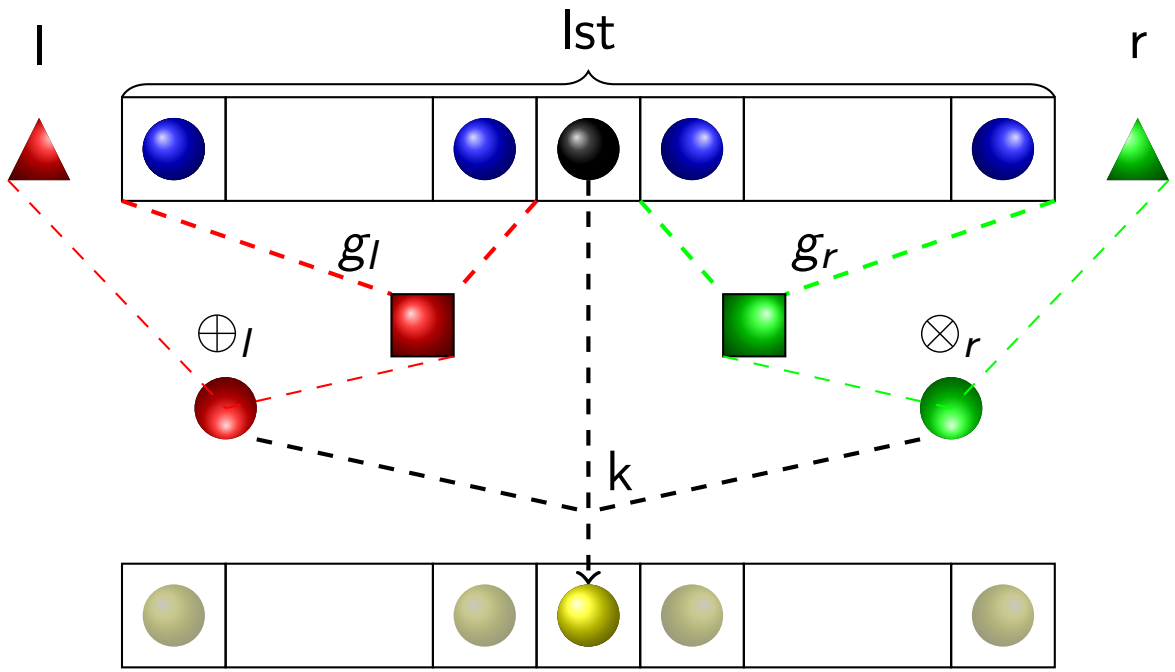
  ==

- *bh_seq*

map3 k
 (*map* g_l (*inits* l))
 l
 (*map* g_r (*tails* l))

- *bh_seq_opt*

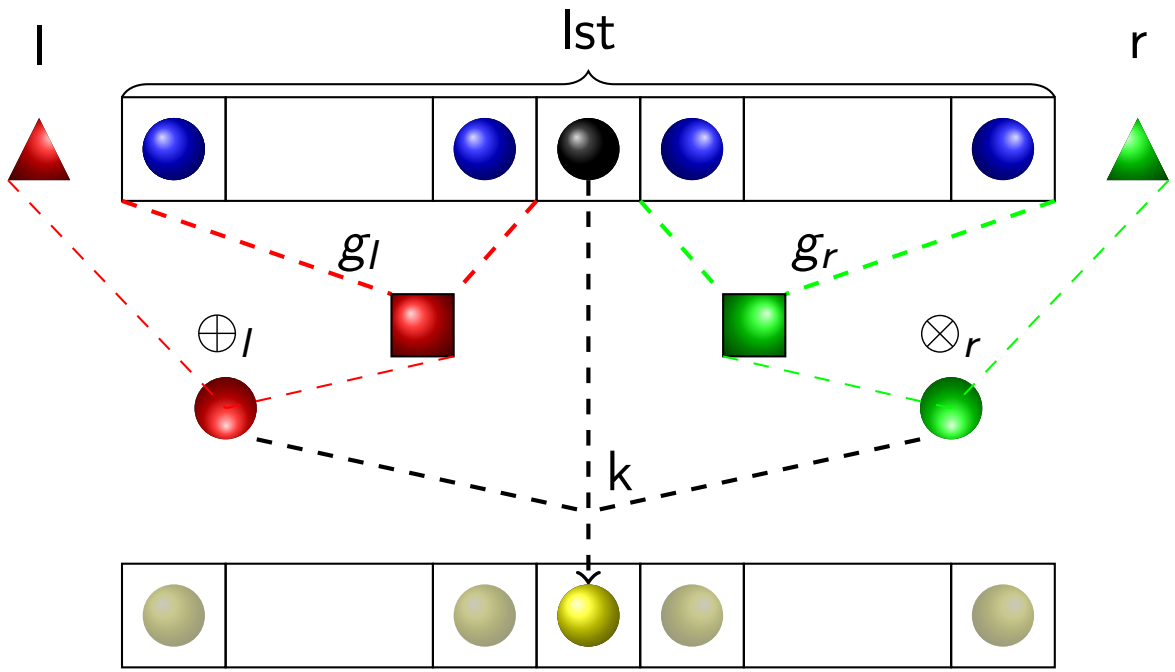
map3 k
 (*fold_left* $\oplus_l$ l)
 l
 (*fold_left* $\otimes_r$ l)



$\forall$ bh *l lst r i*, *nth i* (bh *l lst r*) =
k
(*l* $\oplus_l$(*gl* (*nth i* (*inits lst*))))
(*nth i lst*)
(*gr* (*nth i* (*tails lst*))) $\otimes_r$ *r*)

L'homomorphisme BSP - Calcul séquentiel

- *bh_comp*
  ==
- *bh_seq*
  ==
- *bh_seq_opt*



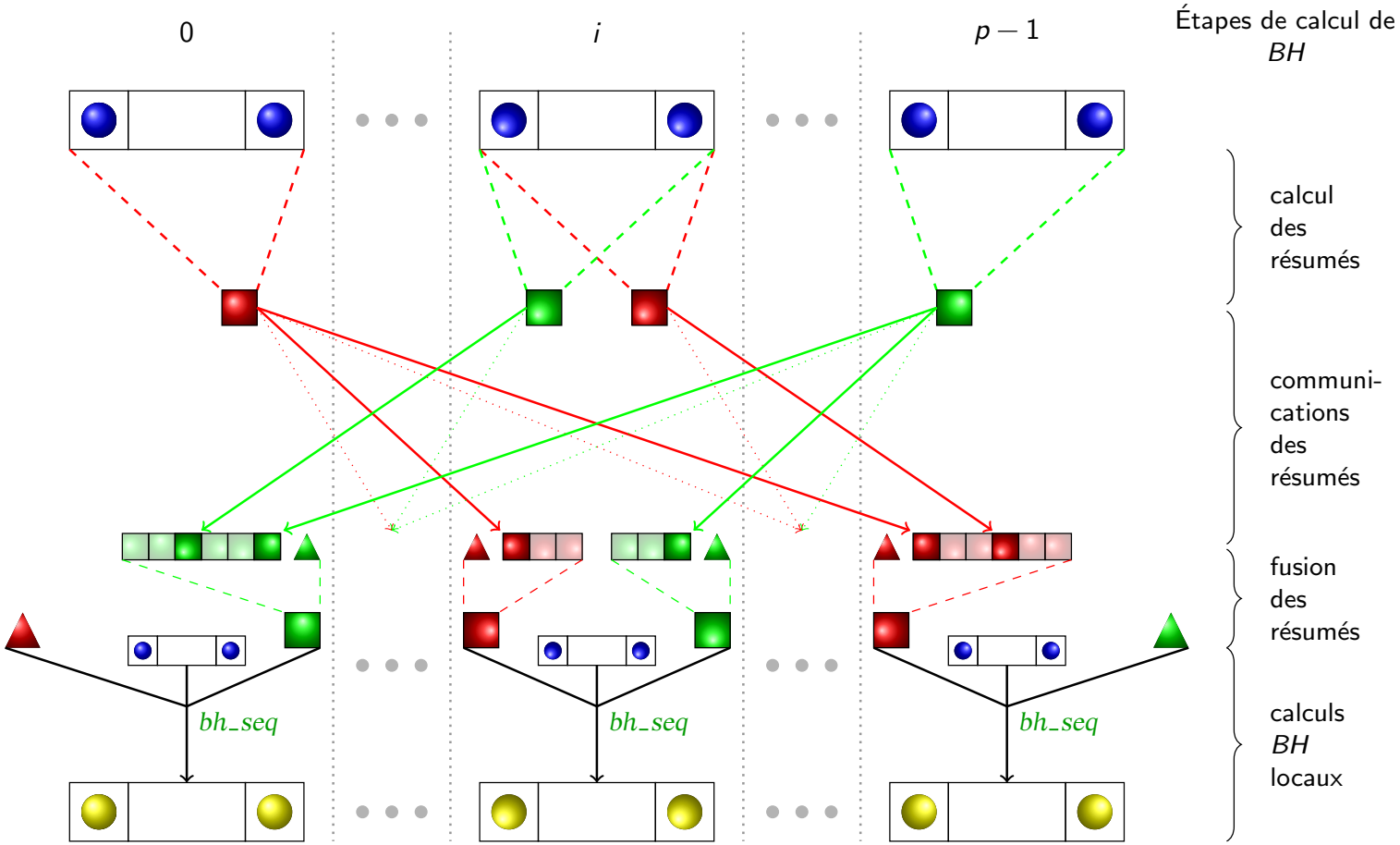
$$\forall\ \text{bh}\ l\ lst\ r\ i,\ nth\ i\ (\text{bh}\ l\ lst\ r) = k\ (l \oplus_l (gl\ (nth\ i\ (inits\ lst))))\ (nth\ i\ lst)\ (gr\ (nth\ i\ (tails\ lst))) \otimes_r r)$$

L'homomorphisme BSP - Modélisation

```
Class BH_PROP (f : list A → L → R → list B) :=
  {
    bh_k : L → A → R → B;
    bh_gl : list A → L;
    bh_opl : L → L → L;
    bh_gr : list A → R;
    bh_opr : R → R → R;
    bh_nil : ∀ l r,  f [] l r = [];
    bh_singleton : ∀ a l r,  f [a] l r = [ bh_k l a r ];
    bh_append : ∀ (x y : list A) l r,
          f (x++y) l r =
          f x l (bh_opr (bh_gr y) r) ++ f y (bh_opl l (bh_gl x)) r
    bh_left_homomorphism : Homomorphism' bh_opl bh_gl;
    bh_right_homomorphism : Homomorphism' bh_opr bh_gr;
  }.
```

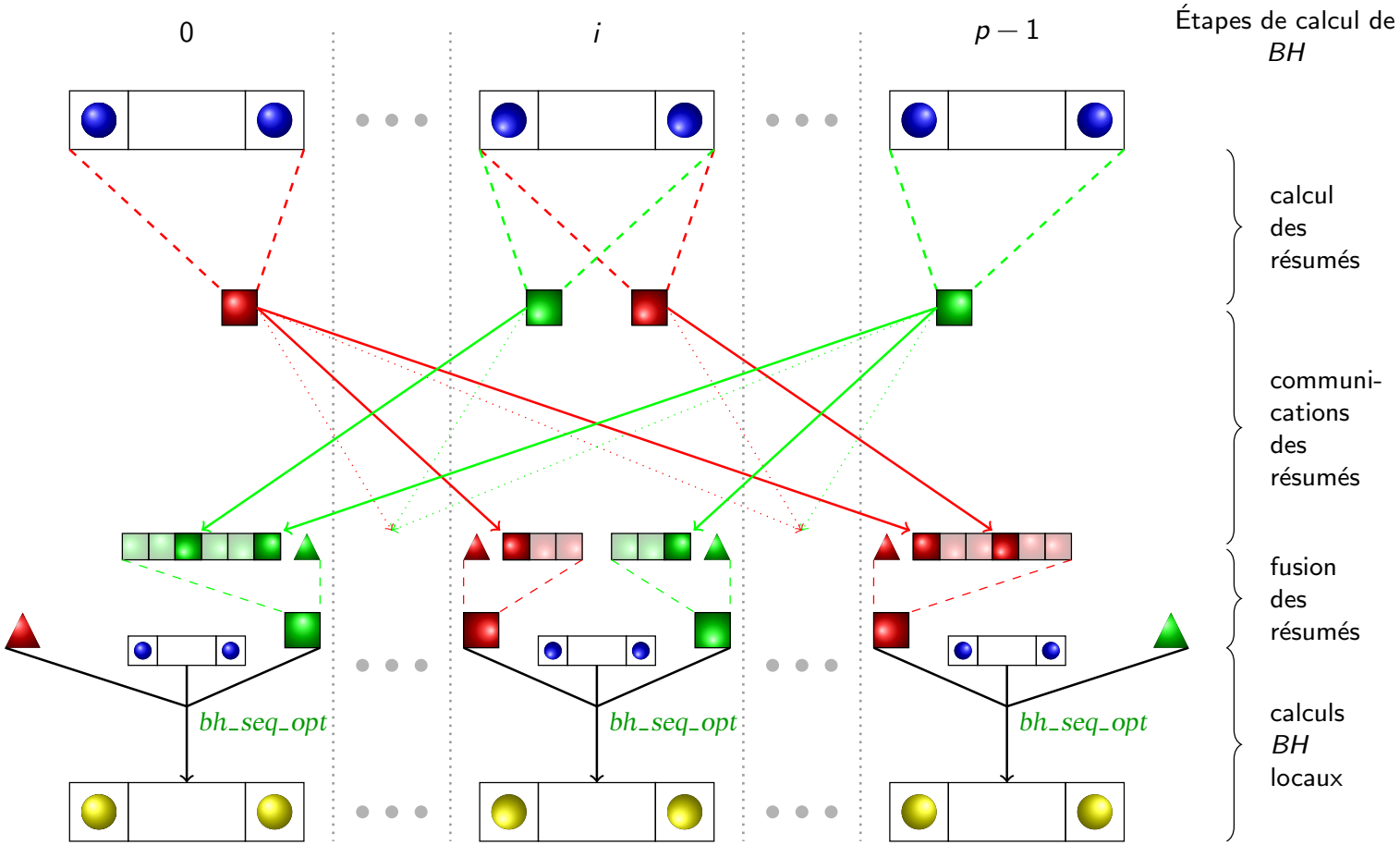
L'homomorphisme BSP - Calcul parallèle

bh_bsml



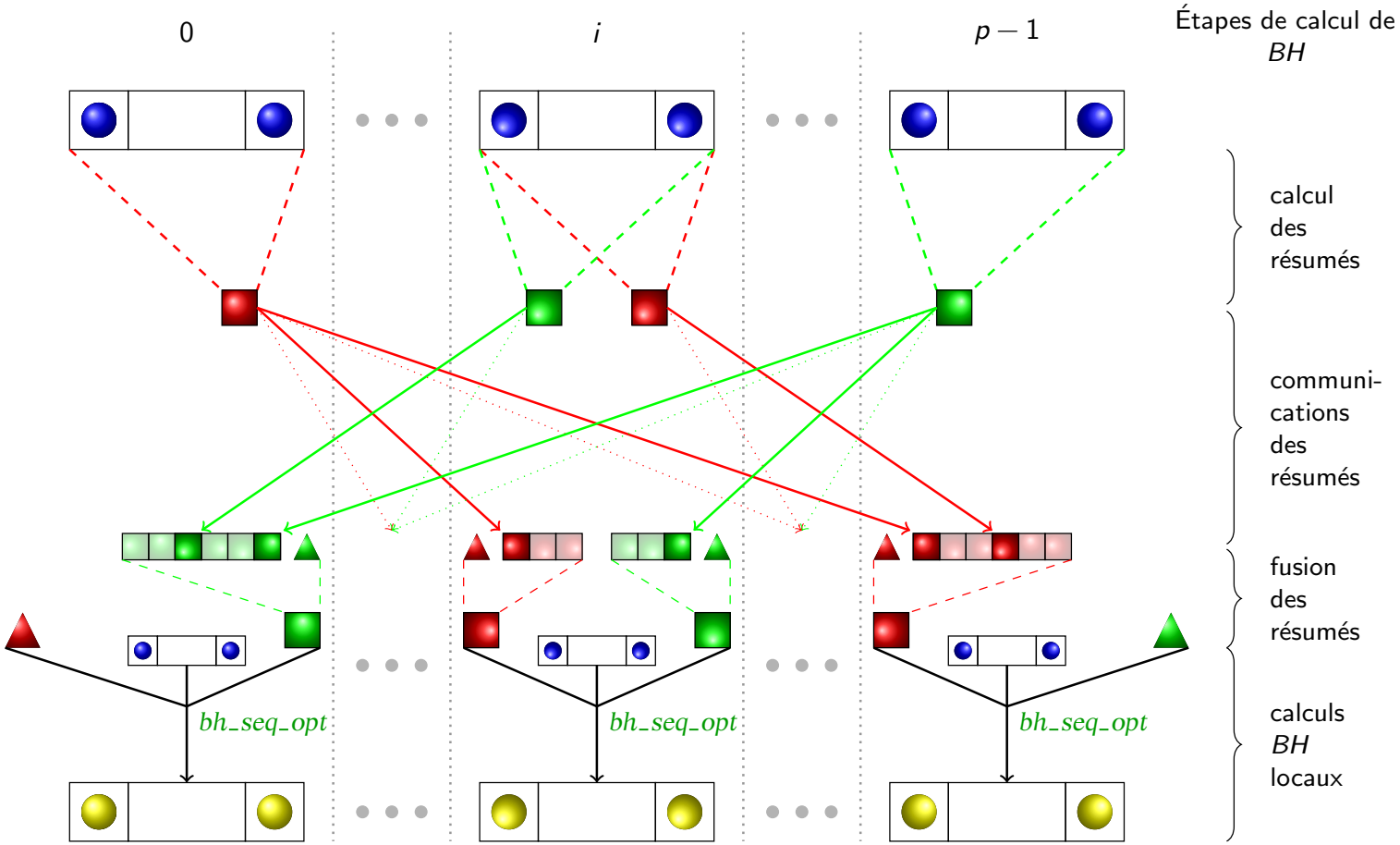
L'homomorphisme BSP - Calcul parallèle

bh_bsml_opt



L'homomorphisme BSP - Calcul parallèle

bh_bsml_opt Parallel (*bh_comp*) (*bh_bsml_opt*)



Apports

	L. Gesbert	J. Tesson
Spécification	*bh_comp*	BH_PROP BH_PROP f $\rightarrow$ f == *bh_comp*
Calcul séquentiel	*bh_seq*	*bh_seq_opt*
Calcul parallèle	*bh_bsml* (version erronée)	*bh_bsml* (version ok) *bh_bsml_opt*
Correction		
Généralité	l= g_l [] r= g_r []	pour tout l et r
Composabilité	X	✓

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ l\ r = & [\,] \\
\text{heat } [a]\ l\ r = & [\text{Formula } a\ l\ r] \\
\text{heat } (x +\!\!+ y)\ l\ r = & \text{heat } x\ l\ (\text{hd } y\ \ r) +\!\!+ \\
 & \quad \text{heat } y\ (\text{last } x\ l)\ r
\end{cases}
$$

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ l\ r = & [\,] \\
\text{bh } [\,]\ l\ r = & [\,] \\
\text{heat } [a]\ l\ r = & [\text{Formula } a\ l\ r] \\
\text{bh } [a]\ l\ r = & [k\ a\ l\ r] \\
\text{heat } (x +\!\!+ y)\ l\ r = & \text{heat } x\ l\ (\text{hd } y\ \ r) +\!\!+ \\
 & \quad \text{heat } y\ (\text{last } x\ l)\ r \\
\text{bh } (x +\!\!+ y)\ l\ r = & \text{bh } x\ l\ (g_r\ y \otimes_r r) +\!\!+ \\
 & \quad \text{bh } y\ (l \oplus_l g_l\ x)\ r
\end{cases}
$$

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ l\ r = & [\,] \\
\text{bh } [\,]\ l\ r = & [\,] \\
\text{heat } [a]\ l\ r = & [\text{Formula } a\ l\ r] \\
\text{bh } [a]\ l\ r = & [k\ a\ l\ r] \\
\text{heat } (x +\!\!+ y)\ l\ r = & \text{heat } x\ l\ (\text{hd } y\ \ r) +\!\!+ \\
 & \quad \text{heat } y\ (\text{last } x\ l)\ r \\
\text{bh } (x +\!\!+ y)\ l\ r = & \text{bh } x\ l\ (g_r\ y \otimes_r r) +\!\!+ \\
 & \quad \text{bh } y\ (l \oplus_l g_l\ x)\ r
\end{cases}
$$

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ l\ r = & [\,] \\
\text{bh } [\,]\ l\ r = & [\,] \\
\text{heat } [a]\ l\ r = & [\text{Formula } a\ l\ r] \\
\text{bh } [a]\ l\ r = & [k\ a\ l\ r] \\
\text{heat } (x +\!\!+ y)\ l\ r = & \text{heat } x\ l\ (\text{hd\_option } y <<\ r) +\!\!+ \\
 & \quad \text{heat } y\ (\text{last\_option } x >>\ l)\ r \\
\text{bh } (x +\!\!+ y)\ l\ r = & \text{bh } x\ l\ (g_r\ y \otimes_r r) +\!\!+ \\
 & \quad \text{bh } y\ (l \oplus_l g_l\ x)\ r
\end{cases}
$$

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{bh } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{heat } [\text{a}]\ \text{l}\ \text{r} = & [\text{Formula a l r}] \\
\text{bh } [\text{a}]\ \text{l}\ \text{r} = & [\text{k a l r}] \\
\text{heat } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{heat x l } (\text{hd\_option y} \ll \text{r}) +\!\!+ \\
 & \text{heat y } (\text{last\_option x} \gg \text{l})\ \text{r} \\
\text{bh } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{bh x l } (g_r\ \text{y} \otimes_r \text{r}) +\!\!+ \\
 & \text{bh y } (\text{l} \oplus_l g_l\ \text{x})\ \text{r}
\end{cases}
$$

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

Parallel (*heatSeq*) (parallel (f := *heatSeqBH* ...))

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ l\ r = & [\,] \\
\text{bh } [\,]\ l\ r = & [\,] \\
\text{heat } [a]\ l\ r = & [\text{Formula } a\ l\ r] \\
\text{bh } [a]\ l\ r = & [k\ a\ l\ r] \\
\text{heat } (x +\!\!+ y)\ l\ r = & \text{heat } x\ l\ (\text{hd\_option } y <<\ r) +\!\!+ \\
 & \quad \text{heat } y\ (\text{last\_option } x >>\ l)\ r \\
\text{bh } (x +\!\!+ y)\ l\ r = & \text{bh } x\ l\ (g_r\ y \otimes_r r) +\!\!+ \\
 & \quad \text{bh } y\ (l \oplus_l g_l\ x)\ r
\end{cases}
$$

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

Parallel (*heatSeq*) (parallel (f := *heatSeqBH* ...))

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{bh } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{heat } [\text{a}]\ \text{l}\ \text{r} = & [\text{Formula a l r}] \\
\text{bh } [\text{a}]\ \text{l}\ \text{r} = & [\text{k a l r}] \\
\text{heat } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{heat x l } (\text{hd\_option y} \ll \text{r}) +\!\!+ \\
 & \text{heat y } (\text{last\_option x} \gg \text{l})\ \text{r} \\
\text{bh } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{bh x l } (g_r\ \text{y} \otimes_r \text{r}) +\!\!+ \\
 & \text{bh y } (\text{l} \oplus_l g_l\ \text{x})\ \text{r}
\end{cases}
$$

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

Parallel (*heatSeq*) (parallel (f := *heatSeqBH* ...))

Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{bh } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{heat } [\text{a}]\ \text{l}\ \text{r} = & [\text{Formula a l r}] \\
\text{bh } [\text{a}]\ \text{l}\ \text{r} = & [\text{k a l r}] \\
\text{heat } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{heat x l } (\text{hd\_option y} \ll \text{r}) +\!\!+ \\
 & \text{heat y } (\text{last\_option x} \gg \text{l})\ \text{r} \\
\text{bh } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{bh x l } (g_r\ \text{y} \otimes_r \text{r}) +\!\!+ \\
 & \text{bh y } (\text{l} \oplus_l g_l\ \text{x})\ \text{r}
\end{cases}
$$

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

Parallel (*heatSeq*) (parallel (f := *heatSeqBH* ...))

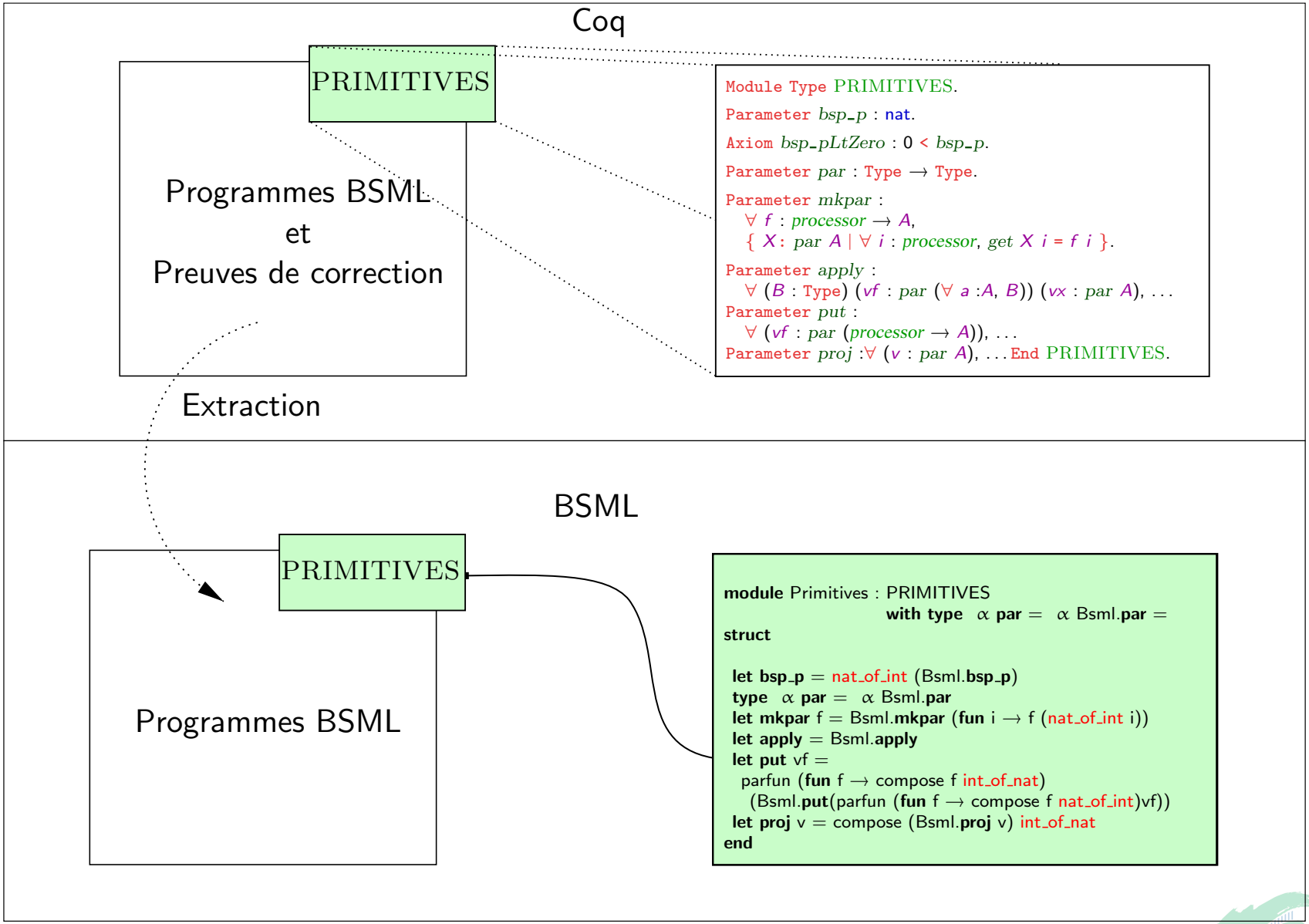
Propriété de heat

$$
\begin{cases}
\text{heat } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{bh } [\,]\ \text{l}\ \text{r} = & [\,] \\
\text{heat } [\text{a}]\ \text{l}\ \text{r} = & [\text{Formula a l r}] \\
\text{bh } [\text{a}]\ \text{l}\ \text{r} = & [\text{k a l r}] \\
\text{heat } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{heat x l } (\text{hd\_option y} \ll \text{r}) +\!\!+ \\
 & \text{heat y } (\text{last\_option x} \gg \text{l})\ \text{r} \\
\text{bh } (\text{x} +\!\!+ \text{y})\ \text{l}\ \text{r} = & \text{bh x l } (g_r\ \text{y} \otimes_r \text{r}) +\!\!+ \\
 & \text{bh y } (\text{l} \oplus_l g_l\ \text{x})\ \text{r}
\end{cases}
$$

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

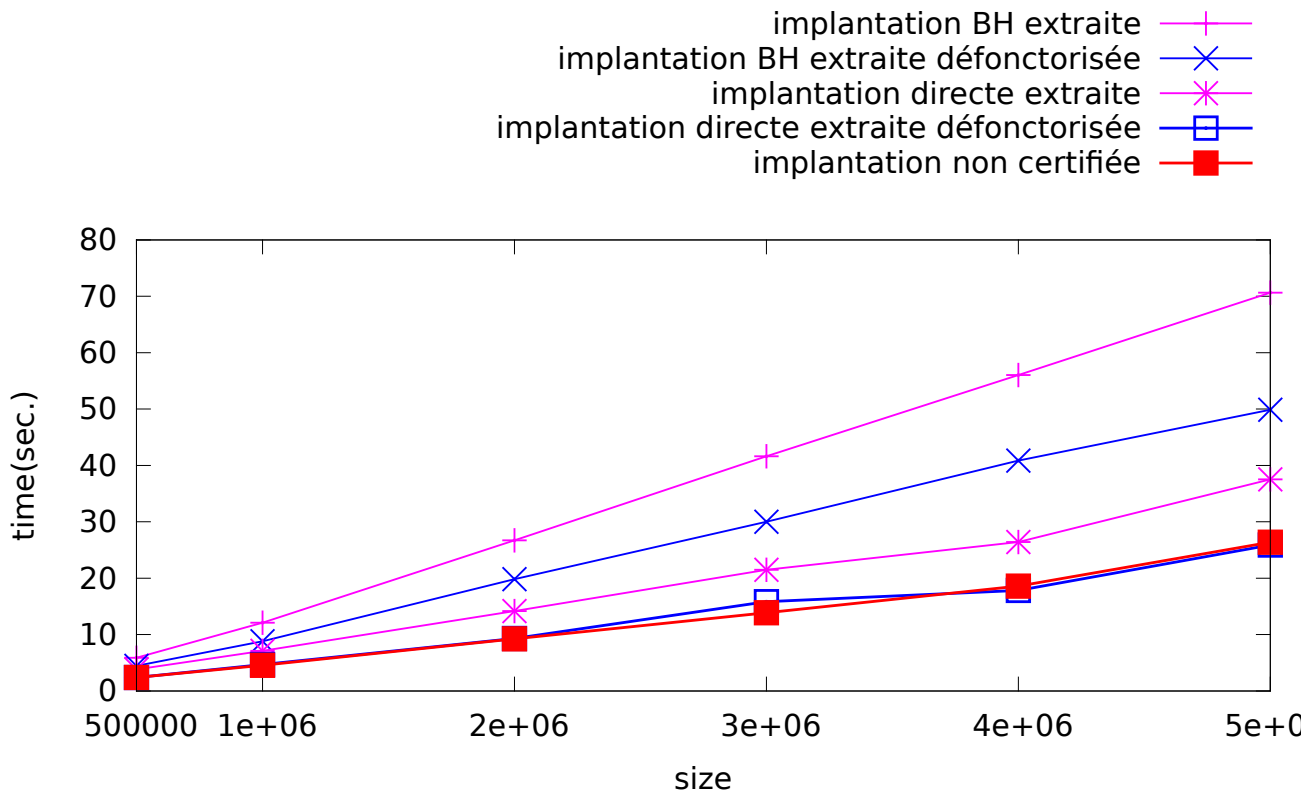
Parallel (*heatSeq*) (parallel (f := *heatSeqBH* ...))

Extraction


Coq
PRIMITIVES
Programmes BSML
et
Preuves de correction
Module Type PRIMITIVES.
Parameter bsp_p : nat.
Axiom bsp_pLtZero : 0 < bsp_p.
Parameter par : Type → Type.
Parameter mkpar :
∀ f : processor → A,
{ X : par A | ∀ i : processor, get X i = f i }.
Parameter apply :
∀ (B : Type) (vf : par (∀ a :A, B)) (vx : par A), ...
Parameter put :
∀ (vf : par (processor → A)), ...
Parameter proj :∀ (v : par A), ... End PRIMITIVES.
Extraction
BSML
PRIMITIVES
Programmes BSML
module Primitives : PRIMITIVES
with type α par = α Bsml.par =
struct
let bsp_p = nat_of_int (Bsml.bsp_p)
type α par = α Bsml.par
let mkpar f = Bsml.mkpar (fun i → f (nat_of_int i))
let apply = Bsml.apply
let put vf =
parfun (fun f → compose f int_of_nat)
(Bsml.put(parfun (fun f → compose f nat_of_int)vf))
let proj v = compose (Bsml.proj v) int_of_nat
end

Expérimentation

- MIReV - grappe de PCs
- 16 processeurs
- Défonctorisation pour permettre au compilateur d'optimiser



Conclusion - http://traclifo.univ-orleans.fr/SDPP

		Spécification	Preuve
Modélisation de BSML	Primitives	**30**	0
	Propriétés & stdlib	216	464
	Implantation séquentielle	60	35
Parallélisation correcte		91	15
	Répartition listes	622	602
Squelettes	BH	456	884
	Autres(map,filter,last,...)	403	226
Applications	Heat equation	199	363
	Heat equation BH	186	**57**
	Comptage	35	**0**
	Construction de tours	105	59
	Maximum prefix sum	110	**0**
Séquentiel	Bibliothèque coq LIFO : lists, vector, algebra	1995	2827
Total		4508	5532

Conclusion

Développement de programmes BSML dans Coq

- Style similaire à la programmation BSML habituelle
- Extraction de programmes parallèles directement utilisable

Preuve de programmes BSML dans Coq

- Définition de programmes fortement spécifiés
- Parallélisation correcte composable

Squelettes algorithmiques

- Simplification du développement (automatisation de la parallélisation)
- Preuves de correction faites une fois pour toutes

Perspectives

Primitives BSML + **Traits impératifs**

Raisonnement sur les coûts

Squelettes Algorithmiques
Nouveaux squelettes

Programmes et preuves de correction

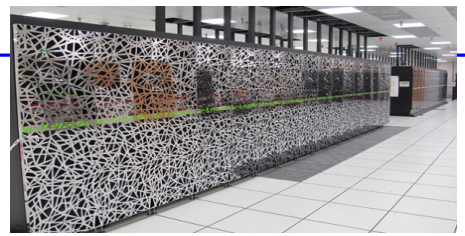
Extraction

Programmes BSML extraits

Implantation des primitives BSML

Défonctorisation et compilation BSML **verifiées**

Exécutions parallèles



Merci

plan

1. Plongement de BSML dans Coq
2. Parallélisation correcte
3. Programmation correcte par squelettes
4. Extraction et expérimentation
5. Conclusion & Perspectives

Annexes

- Évaluation symbolique
- Heat Equation séquentiel
- Heat Equation communication
- Communication : shift
- Parallélisation correcte complet

Exemple - Évaluation symbolique BSML

```
Program Definition Sp : par nat :=
apply
(mkpar (fun p ⇒ (fun i ⇒ i+1)))
(mkpar (fun p ⇒ proj1_sig p)).
```

Exemple - Évaluation symbolique BSML

```
Goal
  ∀ p : processor,
    get Sp p = ‘p + 1.
Proof.
```

Exemple - Évaluation symbolique BSML

```
intros p.
unfold Sp.
```

```
get
 (proj1_sig
  (apply
   (proj1_sig (mkpar (fun (_ : processor) (i : nat) ⇒ i + 1)))
   (proj1_sig (mkpar (fun p0 : processor ⇒ ‘p0)))))) p
=
 ‘p + 1
```

Exemple - Évaluation symbolique BSML

```
rewrite (fun V1 V2 ⇒ proj2_sig (apply V1 V2)).
```

```
(get (proj1_sig (mkpar (fun (_ : processor) (i : nat) ⇒ i + 1))) p)
(get (proj1_sig (mkpar (fun p0 : processor ⇒ ‘p0))) p)
  =
  ‘p + 1
```

Exemple - Évaluation symbolique BSML

```
rewrite (fun f ⇒ proj2_sig (mkpar f)).
```

```
    (fun (i : nat) ⇒ i + 1)
(get (proj1_sig (mkpar (fun p0 : processor ⇒ 'p0))) p)
  =
   'p + 1
```

Exemple - Évaluation symbolique BSML

```
rewrite (fun f ⇒ proj2_sig (mkpar f)).
```

```
(fun (i : nat) ⇒ i + 1)
 ‘p
=
‘p + 1
```

Exemple - Évaluation symbolique BSML

```
  reflexivity.
Qed.
```

```
Unamed_thm is defined
```

```
Fixpoint heatSeq l r dt dx κ (u : list number) : list number :=
      :=
match u with
      | [] ⇒ []
      | ul : : u' ⇒
        match u' with
          | [] ⇒ [ Formula dt dx κ ul l r ]
          | ulPlusOne : : _ ⇒
            (Formula dt dx κ ul l ulPlusOne) : :
            (heatSeq ul r dt dx κ u')
        end
    end.
```

getBounds

```
Program Definition getBounds (A :Type)(l r : A)
  (v : par(list A))(H : ∀ i, get v i ≠ nil) :

  { vr : par A | ∀ (i :processor),
    get vr i =
      if ( i == firstProc ) then l else sLast (get v (i-1))
  } ×
  { vr : par A | ∀ (i :processor),
    get vr i =
      if ( i == lastProc) then r else
                        sHead (get v (min (i+1) lastProc)) }
   :=
    let tmp := getBoundsAux l r H in
    ( parfun (@noSome A) (parSig (fst tmp) _ _) ,
      parfun (@noSome A) (parSig (snd tmp) _ _) ).
```

getboundAux

```
Program Definition getBoundsAux (A :Type)(l r : A)
    (v : par(list A))(H : ∀ i, get v i ≠ nil) :
    { vr : par (option A) | ∀ (i :processor),
      get vr i = Some ( if ( i == firstProc ) then l else
sLast (get v (i-1)) ) } ×
    { vr : par (option A) | ∀ (i :processor),
      get vr i = Some ( if ( i == lastProc) then r else
sHead (get v (min (i+1) lastProc))) } :=
    let msg := put(apply(mkpar(fun (pid :processor) data
(dst :processor) ⇒
      if ( dst == (pid+1) ) && negb(pid == (bsp_p-1)) then
Some (sLast data)
        else if ( dst == (pid-1) ) && (negb(pid == 0)) then
Some (sHead data)
          else None)) (parSig v _ H) ) in
    ( applyat firstProc (constantFunPar processor (Some l))
msg (parSig (mkpar(fun pid⇒pid-1)) _ _ ) ,
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i = get vr ( (i+dec) mod bsp_p) } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p) ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i = get vr ( (i+dec) mod bsp_p) } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p) ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

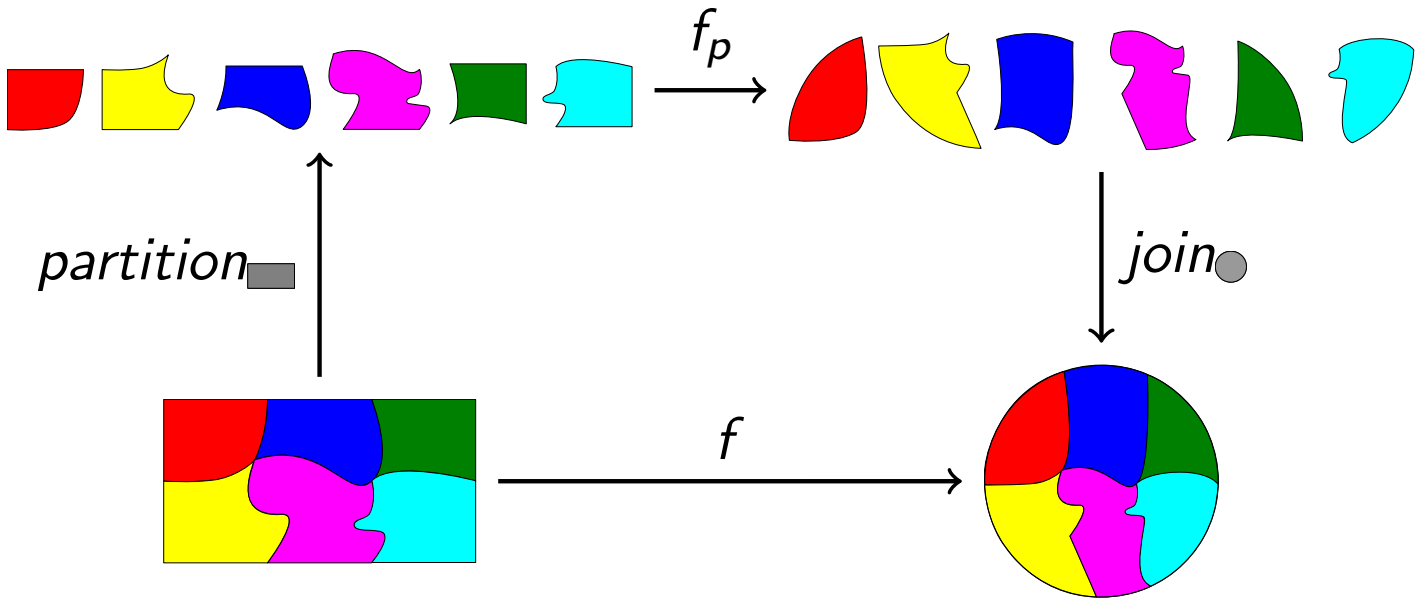
```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

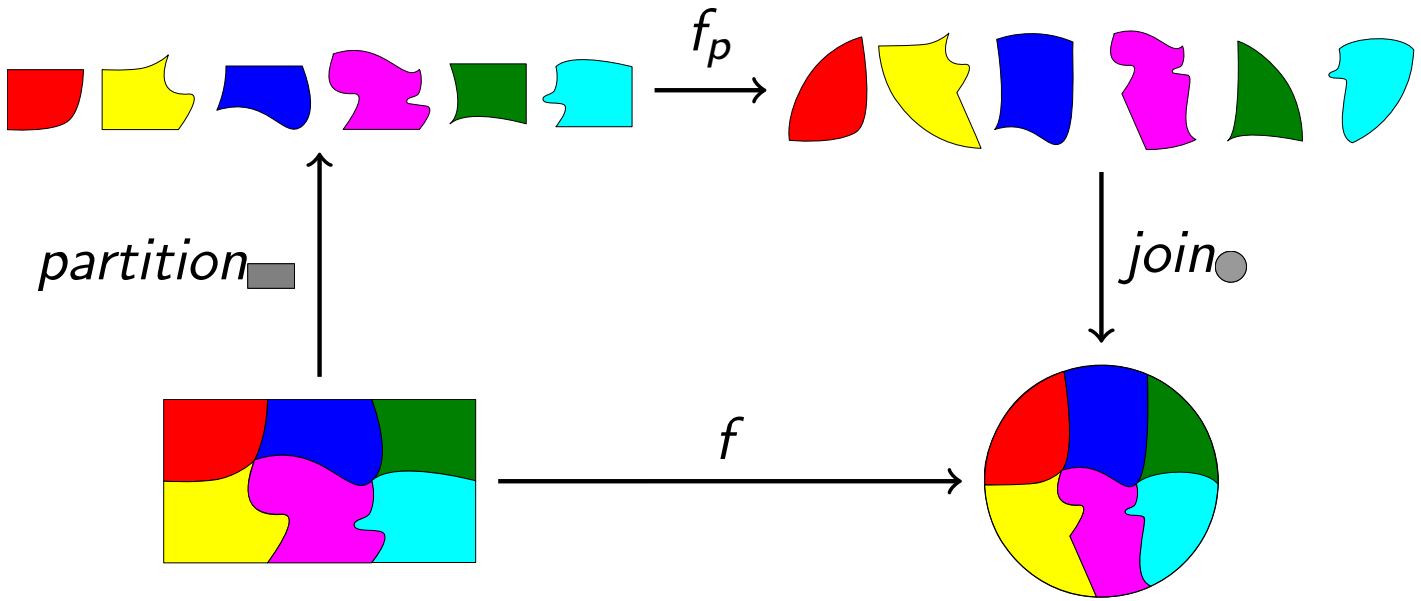
```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i = get vr ( (i+dec) mod bsp_p) } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p) ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i :processor) (f : processor → _) ⇒
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a ⇒ a ≠ None) _ ) .
Next Obligation. ...
```

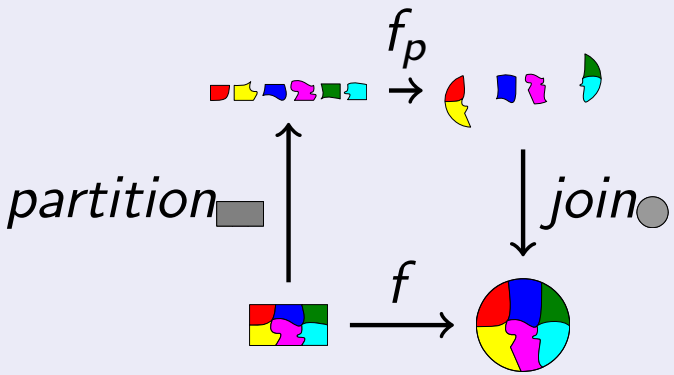
Parallélisation correcte

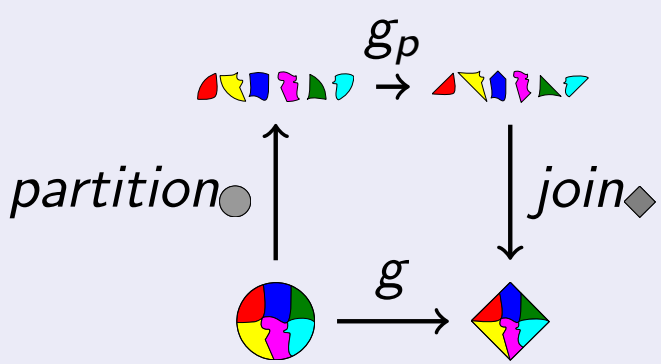


Parallélisation correcte

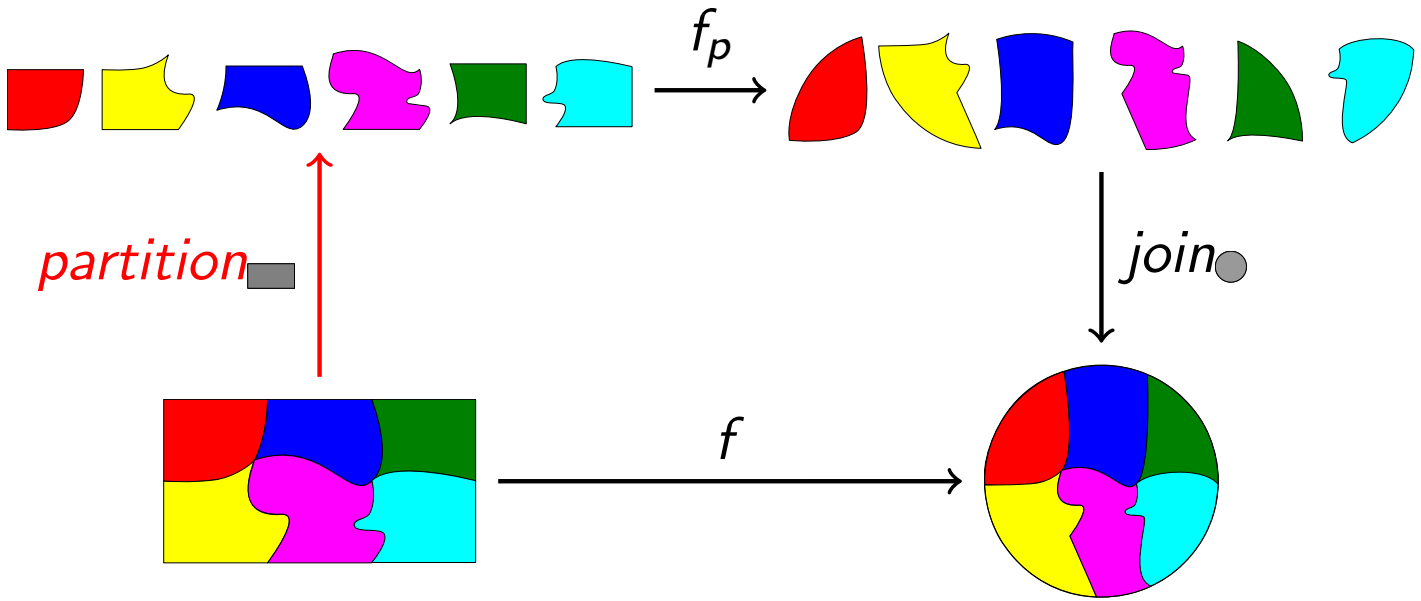


Composabilité : $g_p \circ f_p$?



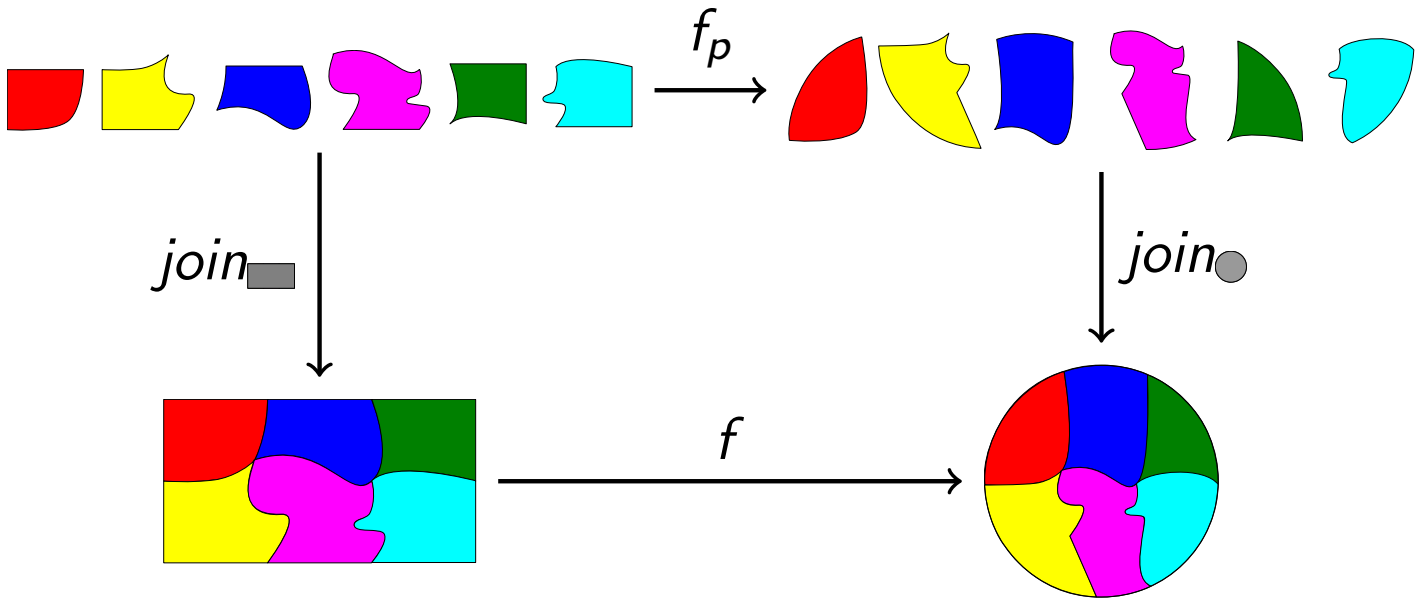


Parallélisation correcte

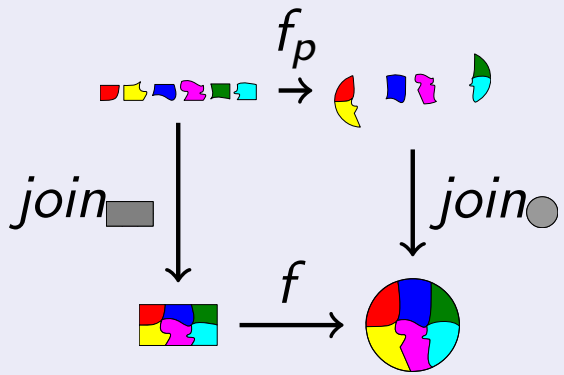


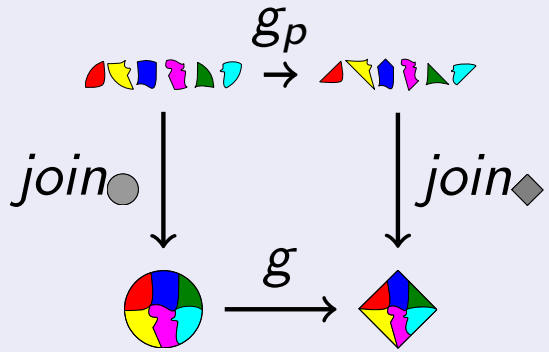
Composabilité : $g_p \circ f_p$?

Parallélisation correcte

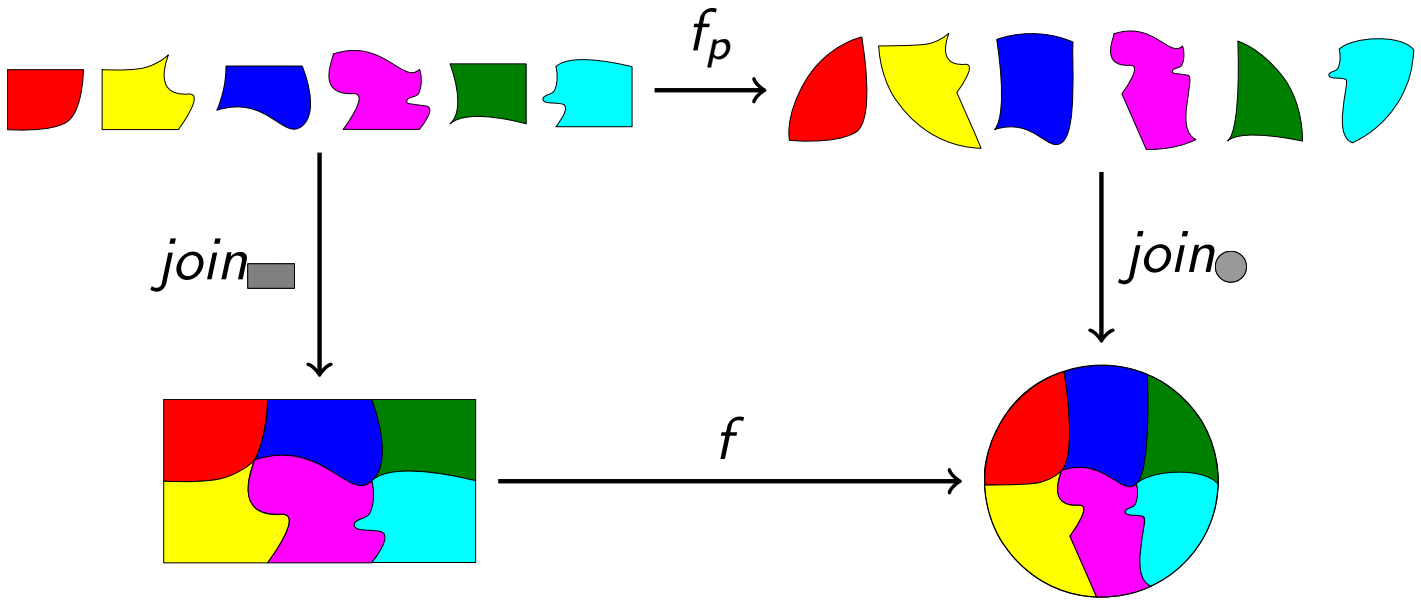


Composabilité : $g_p \circ f_p$?



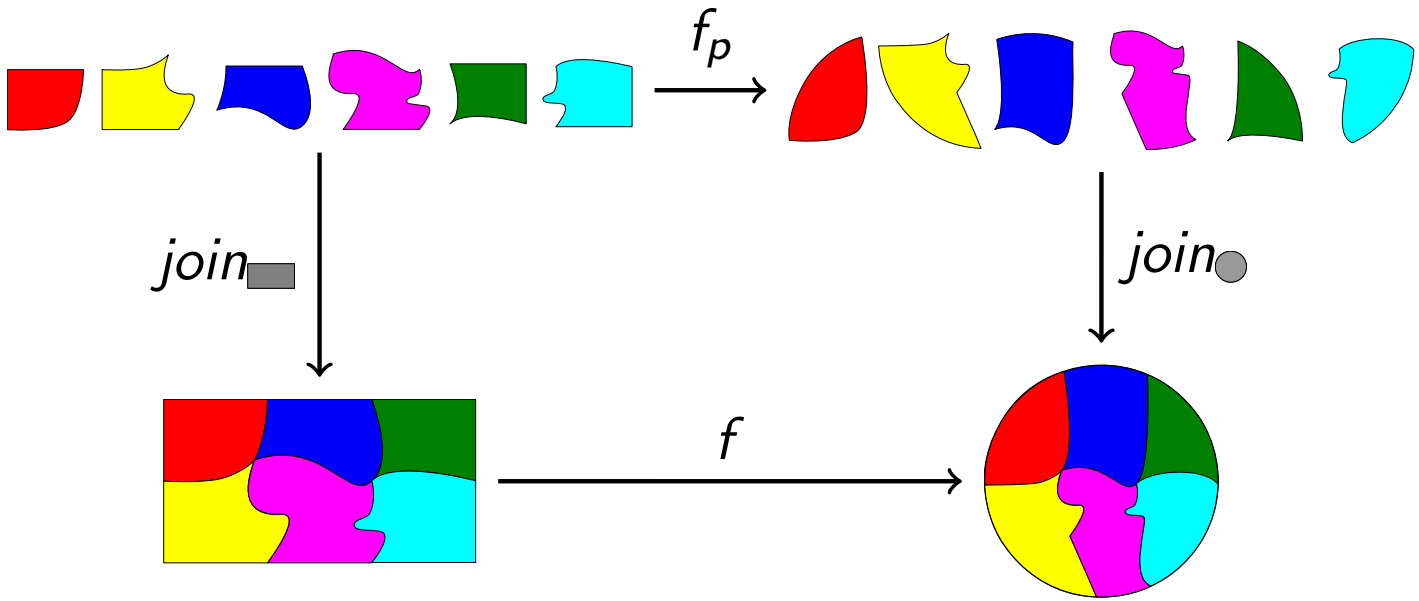


Parallélisation correcte

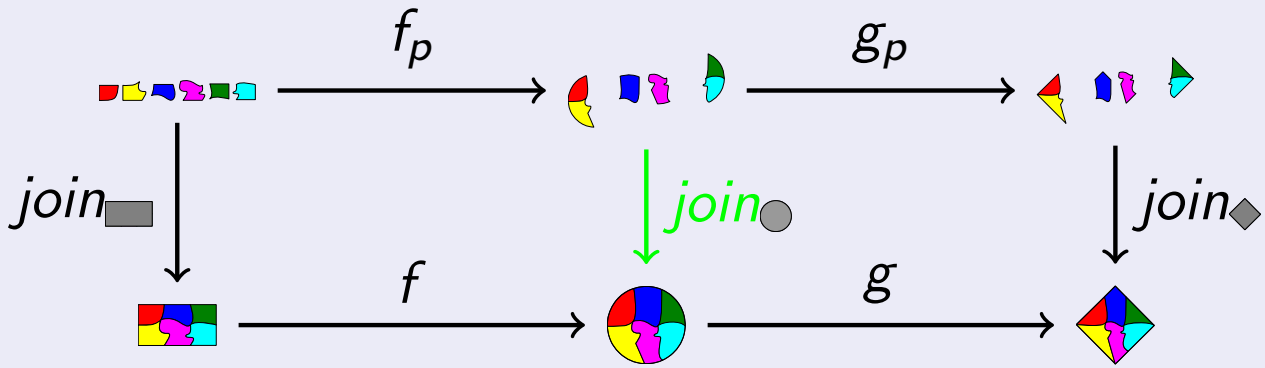


Composabilité : $g_p \circ f_p$?

Parallélisation correcte



Parallélisation correcte composable $g_p \circ f_p$



Fin