

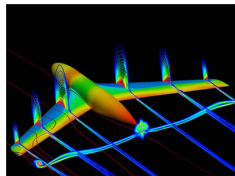
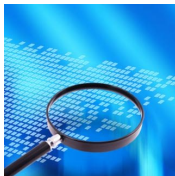
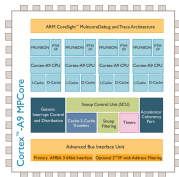
Environnement pour le développement et la preuve de correction systématiques de programmes parallèles fonctionnels

JULIEN TESSON

sous la direction de FRÉDÉRIC LOULERGUE

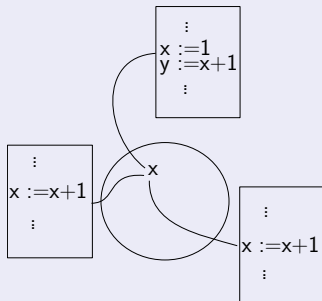
LIFO, Université d'Orléans

Soutenance de Thèse,
8 novembre 2011
Orléans

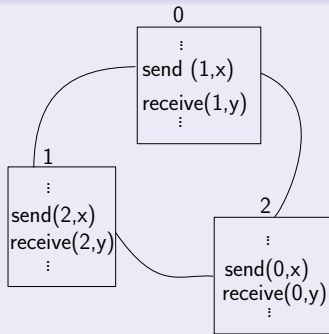


Programmation parallèle

Mémoire partagée



Mémoire distribuée



Modèles bas niveaux

- + Expressif
- Expressif $\rightarrow$ 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

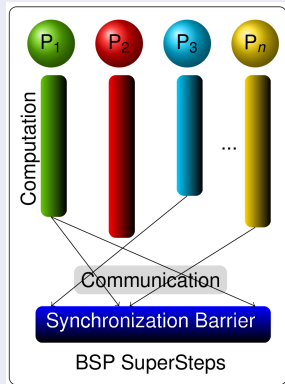
Trois modèles

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

Calculateur BSP

- p paires mémoire/processeur
(de vitesse r)
- Un réseau 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

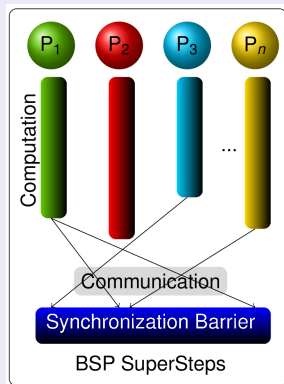
- ▶ Un ensemble de primitives parallèles restreint
- ▶ Universel pour le parallélisme BSP
- ▶ Vue globale du programme
- ▶ Une sémantique formelle simple

BSML

Un langage fonctionnel

- + une structure de données parallèle (sans imbrication)
- + des opérations parallèles sur cette structure

Modèle d'exécution BSP



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

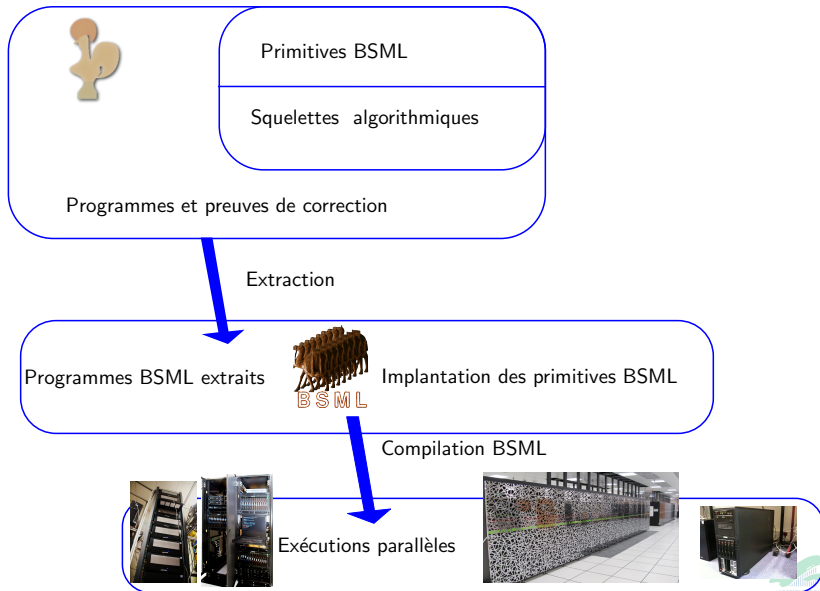
Utiliser les squelettes algorithmiques pour écrire des programmes parallèles corrects par construction.



Nos moyens

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

Vue d'ensemble



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

Preuve de programme dans un assistant de preuve

Plongement profond

- ▶ Arbre de syntaxe abstraite
- + règles sémantique

Plongement superficiel

- ▶ Langage de l'assistant de preuve
- = langage de programmation
- + axiomatisation

Programme

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

Lemme

```
Lemma sumSpec :  
   $\forall n, \text{sum } n = n * (1 + n) / 2.$   
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  $\Rightarrow$  0  
    | S n'  $\Rightarrow$  n + sum' n'  
  end.
```

Next Obligation.
:
omega.
Defined.

Programme

```
Fixpoint sum (n : nat) : nat :=  
  match n with  
    0  $\Rightarrow$  0  
    | S n'  $\Rightarrow$  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  
| 0 → 0  
| S n' → plus n (sum n')
```

Plongement de BSMML 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 :

v_0	v_1	$\dots$	v_{p-1}
-------	-------	---------	-----------

Module Type PRIMITIVES.

Parameter bsp_p : nat.

Axiom $bsp_pLtZero$: $0 < bsp_p$.

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

Parameter par : Type $\rightarrow$ Type.

⋮

Création d'un vecteur parallèle

► **mkpar** : $(\text{int} \rightarrow 'a) \rightarrow 'a \text{ par}$

(**mkpar** *f*)

(<i>f</i> 0)	(<i>f</i> 1)	...	(<i>f</i> (<i>p</i> - 1))
---------------	---------------	-----	-----------------------------

⋮

Parameter *get* : $\forall A : \text{Type}, \text{par } A \rightarrow \text{processor} \rightarrow A$.

Parameter *mkpar* : $\forall (A : \text{Type}),$

$\forall (f : \text{processor} \rightarrow A),$

$\{ X : \text{par } A \mid \forall i : \text{processor}, \text{get } X \ i = f \ i \}.$

⋮

Application parallèle point à point

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

$$\begin{aligned}
 & \left(\text{apply} \begin{array}{|c|c|c|c|} \hline f_0 & f_1 & \dots & f_{p-1} \\ \hline v_0 & v_1 & \dots & v_{p-1} \\ \hline \end{array} \right) \\
 = & \begin{array}{|c|c|c|c|} \hline (f_0 \ v_0) & (f_1 \ v_1) & \dots & (f_{p-1} \ v_{p-1}) \\ \hline \end{array}
 \end{aligned}$$

⋮

Parameter *apply* : $\forall (A \ B : \text{Type}),$
 $\forall (vf : \text{par } (A \rightarrow B)) (vx : \text{par } A),$
 $\{ X : \text{par } B \mid \forall i : \text{processor},$
 $\text{get } X \ i = (\text{get } vf \ i) (\text{get } vx \ i) \}.$

⋮

Communications

► **put**: $(\text{int} \rightarrow 'a) \text{ par} \rightarrow (\text{int} \rightarrow 'a) \text{ par}$

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

$$g_i j = f_j i$$

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** :

$$\begin{aligned} &\forall (vf : \text{processor} \rightarrow A), \\ &\quad \{ X : \text{processor} \rightarrow A \mid \\ &\quad \quad \forall i j : \text{processor}, \text{get } X \ i \ j = \text{get } vf \ j \ i \}. \end{aligned}$$

Communications - du local au global

► **proj**: 'a par $\rightarrow$ (int $\rightarrow$ 'a)

$$\left(\text{proj} \quad \boxed{v_0 \quad v_1 \quad \dots \quad v_{p-1}} \right) = \begin{array}{ccc} \text{function} & 0 & \rightarrow v_0 \\ & 1 & \rightarrow v_1 \\ & \vdots & \\ & p-1 & \rightarrow v_{p-1} \end{array}$$

⋮

Parameter *proj* :

$\forall (v : \text{par } A),$

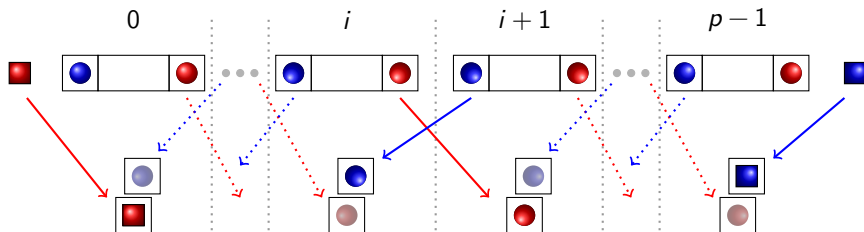
$\{ X : \text{processor} \rightarrow A \mid \forall i : \text{processor}, X\ i = \text{get } v\ i \}.$

End PRIMITIVES.

	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 nécessite une retouche du code extrait	X	✓
Réalisation séquentielle	Utilise axiomes		✓

Exemple - Spécification forte d'une fonction

Program Definition *getBounds* ($A : \text{Type}$) ($r : A$)
($v : \text{par}(\text{list } A)$) ($H : \forall i, \text{get } v \ i \neq \text{nil}$) :
{ $vl : \text{par } A \mid \forall (i : \text{processor}), \text{get } vl \ i =$
 $\text{if } (i == \text{firstProc}) \text{ then } l \text{ else } s\text{Last } (\text{get } v \ (i-1))$ } $\times$
{ $vr : \text{par } A \mid \forall (i : \text{processor}), \text{get } vr \ i =$
 $\text{if } (i == \text{lastProc}) \text{ then } r \text{ else}$
 $s\text{Head } (\text{get } v \ (\text{min } (i+1) \ \text{lastProc}))$ }



Exemple - Spécification forte d'une fonction

Program Definition *getBounds* (*A* : Type) (*l r* : *A*)
(*v* : *par*(*list A*))(*H* : $\forall i, \text{get } v \ i \neq \text{nil}$) :
{ *vl* : *par A* | $\forall (i : \text{processor}), \text{get } vl \ i =$
 if (*i* == *firstProc*) *then l* *else sLast* (*get v (i-1)*) } $\times$
{ *vr* : *par A* | $\forall (i : \text{processor}), \text{get } vr \ i =$
 if (*i* == *lastProc*) *then r* *else*
 sHead (*get v (min (i+1) lastProc)*) }



- ① Plongement de BSML dans Coq
- ② Parallélisation correcte
 - Répartition des données
 - Parallélisation correcte
 - Modélisation en coq
 - Exemple : diffusion de la chaleur
- ③ Programmation correcte par squelettes
- ④ Extraction et expérimentation
- ⑤ Conclusion & Perspectives

Répartition des données

Structure de données partitionnable



Propriétés

$$\text{partition} \circ \text{join} =$$

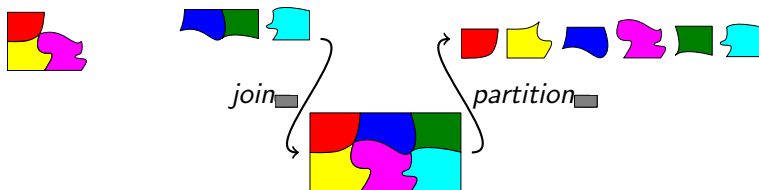
Répartition des données

Structure de données partitionnable



Propriétés

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



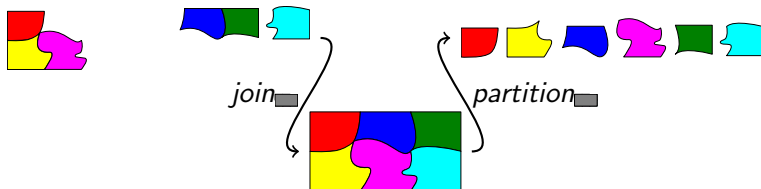
Répartition des données

Structure de données partitionnable



Propriétés

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



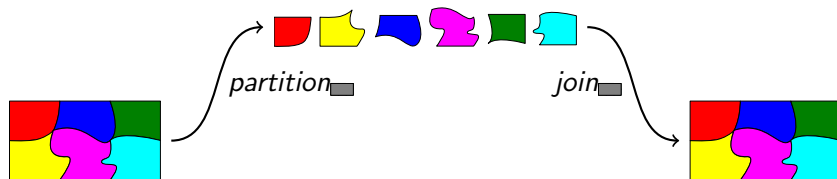
Répartition des données

Structure de données partitionnable



Propriétés

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



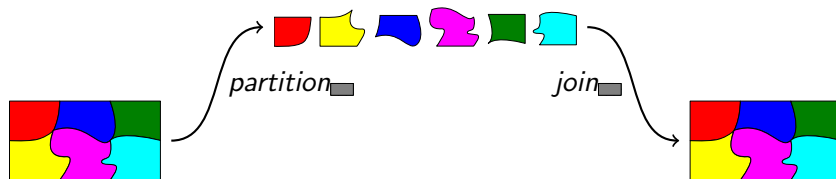
Répartition des données

Structure de données partitionnable

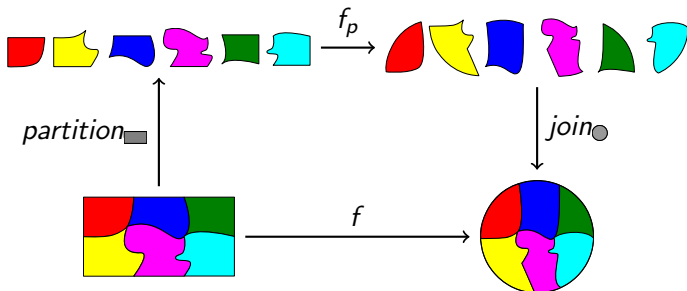


Propriétés

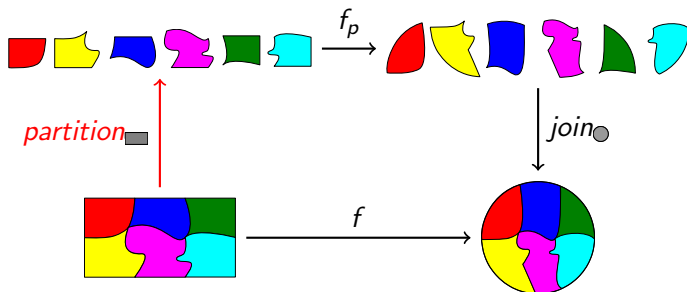
$$join_{\square} \circ partition_{\square} = id_{\square}$$



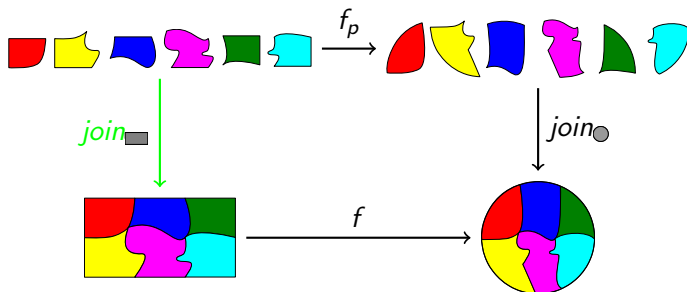
Parallélisation correcte



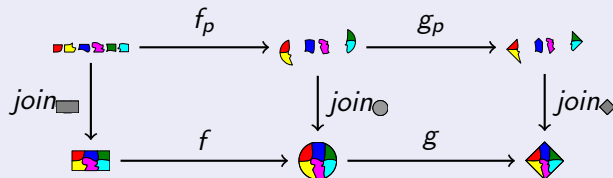
Parallélisation correcte



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, $a \times b \times c + b + c = a \times (b \times c) + (b + c)$.

intros.

replace ($a \times b \times c$) with ($a \times (b \times c)$) by (apply *associative*).

replace ($a \times (b \times c) + b + c$) with ($a \times (b \times c) + (b + c)$)

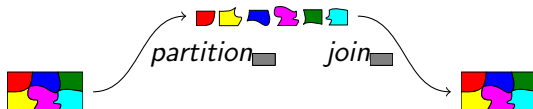
by (apply *associative*).

reflexivity.

Qed.

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 )  
}.
```



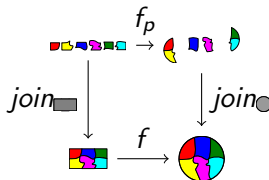
Parallélisation correcte composable - Cog I

Parallélisation correcte

```

Class Parallel '{  $A\_Part$  : Partitionnable  $A$  }
  '{  $B\_Part$  : Partitionnable  $B$  }
  ( $f$  :  $A \rightarrow B$ )
  ( $fp$  :  $parallel\_type(A := A) \rightarrow parallel\_type(A := B)$ ) :=
{
   $parallel\_spec\_match$  :  $\forall par\_a,$ 
     $join (fp (par\_a)) = f (join par\_a)$ 
}.

```



Instance

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

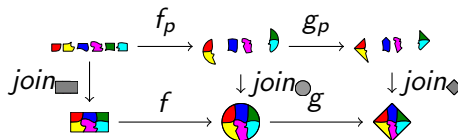
```

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 ∘ g) (fp ∘ gp).
```



Diffusion de la chaleur 1D



Équation de diffusion :

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

- ▶ 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 :

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

Diffusion de la chaleur 1D



Une version discrétisée :

$$\begin{aligned} & (\text{step } \kappa \, dx \, dt \mid r \, u) (x) \\ &= \\ & \quad \frac{\kappa dt}{dx^2} \times (\\ & \quad \quad u(x + dx) \\ & \quad \quad + \quad u(x - dx) \\ & \quad \quad - \quad 2 \times u(x)) \\ & \quad + u(x) \end{aligned}$$

Diffusion de la chaleur 1D



Spécification :

$$\begin{aligned} & (\text{step } \kappa \, dx \, dt \mid r \, u) \, [i] \\ & = \\ & \quad \frac{\kappa dt}{dx^2} \times \left(\begin{aligned} & u[i + 1] \\ & + u[i - 1] \\ & - 2 \times u[i] \end{aligned} \right) \\ & \quad + u[i] \end{aligned}$$

Diffusion de la chaleur 1D



Spécification :

$$\begin{aligned} & (\text{step } \kappa \, dx \, dt \mid r \, u) \, [i] \\ & = \\ & \quad \frac{\kappa dt}{dx^2} \times (\\ & \quad \quad \text{if } i \leq nb_elem \text{ then } r \text{ else } u[i + 1] \\ & \quad \quad + \text{ if } i = 0 \text{ then } l \text{ else } u[i - 1] \\ & \quad \quad - 2 \times u[i]) \\ & \quad + u[i] \end{aligned}$$

Diffusion de la chaleur 1D



Spécification :

```
nth i (step κ dx dt l r u) d
=
   $\frac{\kappa dt}{dx^2} \times ($ 
    if  $i \leq nb\_elem$  then  $r$  else  $nth (i + 1) u d$ 
    + if  $i = 0$  then  $l$  else  $nth (i - 1) u d$ 
    -  $2 \times (nth i u d)$ 
    +  $(nth i u d)$ 
```


Diffusion de la chaleur 1D



Spécification :

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

Diffusion de la chaleur 1D



Spécification :

Definition *StepSpecification* *step* : Prop :=

$\forall (u : \text{list number})(Hu : u \neq []) (l\ r\ dt\ dx\ \kappa : \text{number})$
 $(i : \text{nat})(Hi : i < \text{length } u)(d : \text{number}),$

nth *i* (*step* $\kappa\ dt\ dx\ l\ r\ u$) *d*

=

$\kappa \times dt / (dx \times dx) ($
 $\text{nth } (i+1) \ u \ r$
 + $\text{if beq_nat } i\ 0 \text{ then } l \text{ else } \text{nth } (i-1) \ u \ d$
 - $(\text{nth } i \ u \ d) + (\text{nth } i \ u \ d)$
 + $(\text{nth } i \ u \ d)$
 $)$

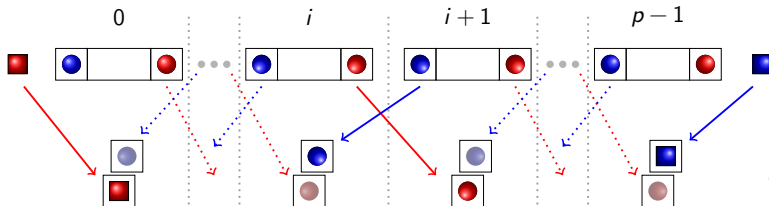
Diffusion de la chaleur 1D

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* : $\forall i, \text{get } u \ i \neq \text{nil}$)
: *par(list number)* :=
let *bounds* := *getBounds* *l r Hu* in
apply (*parfun2*
(*fun l r* $\Rightarrow$ *heatSeq* *l r dt dx κ*) (*fst bounds*) (*snd bounds*))
u.



Correction de l'implantation séquentielle

Lemma *stepSeqFollowsHeatEquationStepSpecification :*
heatEquationStepSpecification heatSeq.

Proof.

⋮

induction *u*

⋮

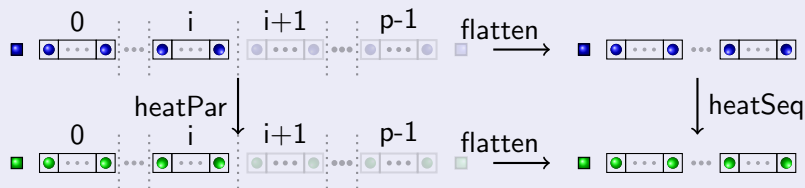
induction *i*

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

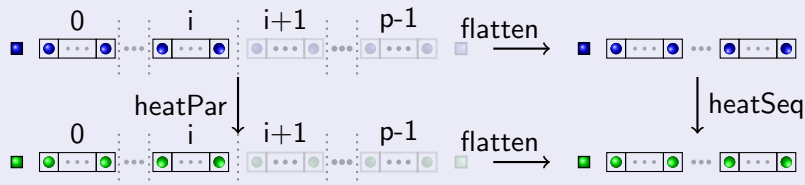


Théorème

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 $heatSeq$ $heatPar$

- ① Plongement de BSML dans Coq
- ② Parallélisation correcte
- ③ Programmation correcte par squelettes
 - Parallélisation automatique
 - Homomorphismes BSP
 - Exemple : diffusion de la chaleur
- ④ Extraction et expérimentation
- ⑤ Conclusion & Perspectives

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*)

|

Parallelizable (*counting A size*)

Parallélisation automatique

parallel (*f* := *counting A size*)

|

parallel_parallelizable

|

Parallel (*counting A size*) (?)

Parallélisation automatique

parallel (*f* := *counting A size*)

|

parallel_parallelizable

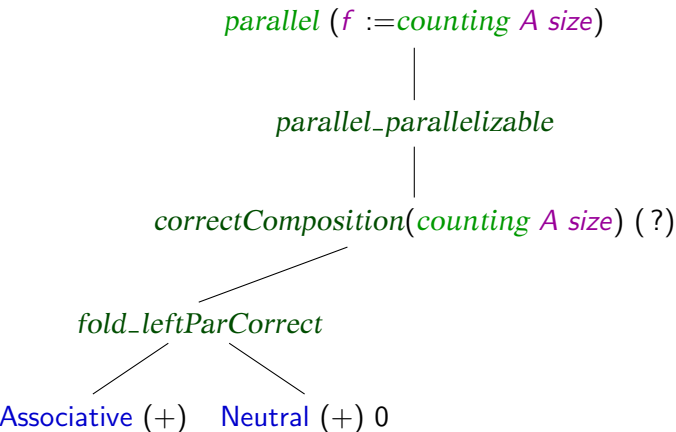
|

correctComposition(*counting A size*) (?)

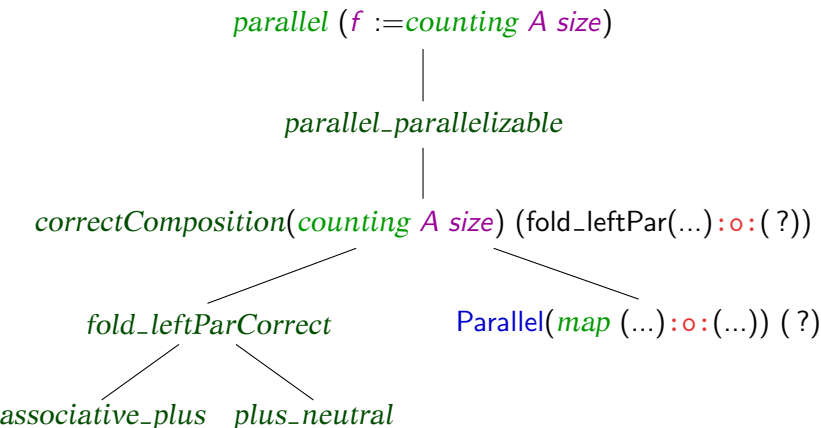
↙

Parallel (*fun* *l* ⇒ *fold_left* (+) *l* 0) (?)

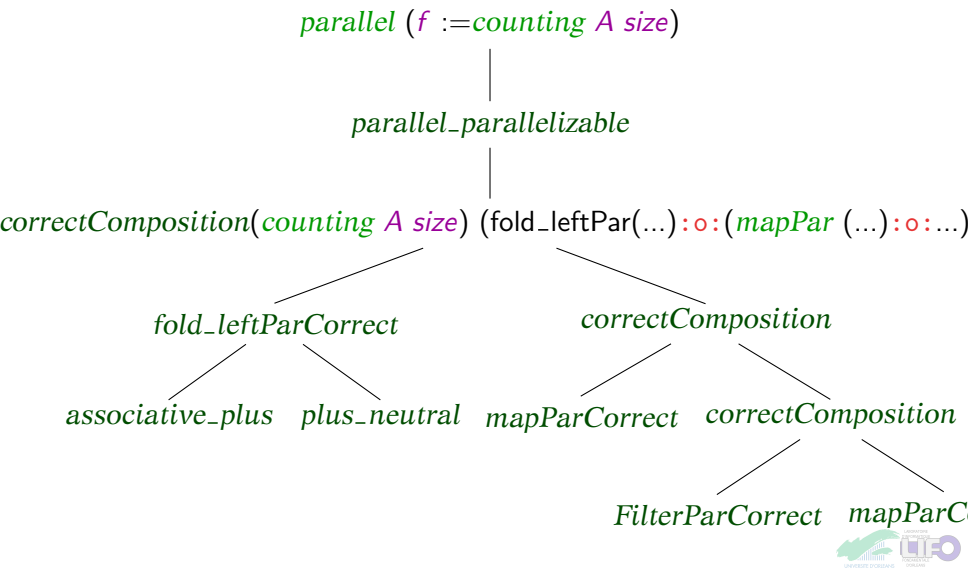
Parallélisation automatique



Parallélisation automatique



Parallélisation automatique

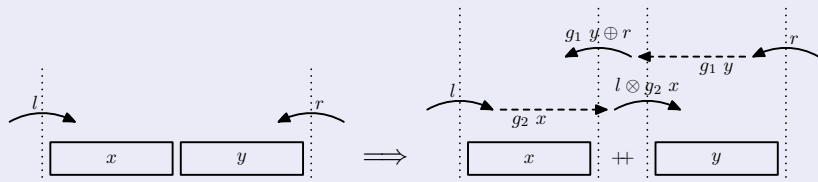


Homomorphismes de listes pour le parallélisme

Homomorphisme de listes

$$h(\boxed{x} ++ \boxed{y}) = \vdots h(\boxed{x}) \odot h(\boxed{y}) \vdots$$

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 :

$$\left\{ \begin{array}{ll} bh [] \mid r = [] & (BH\text{-}nil) \\ bh [a] \mid r = [k \ a \mid r] & (BH\text{-}Singleton) \\ bh (x ++ y) \mid r = bh \ x \mid (g_r \ y \otimes_r \ r) ++ \\ \quad bh \ y \mid (I \oplus_I g_l \ x) \ r & (BH\text{-}Concat) \end{array} \right.$$

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

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

$(g_r (y ++ z)) \otimes_r r = (g_r y) \otimes_r ((g_r z) \otimes_r r)$ (g_r -Concat-faible)

$I \oplus_I (g_l (x ++ y)) = (I \oplus_I (g_l x)) \oplus_I (g_l y)$ (g_l -Concat-faible)

L'homomorphisme BSP : formellement

Définition (BH)

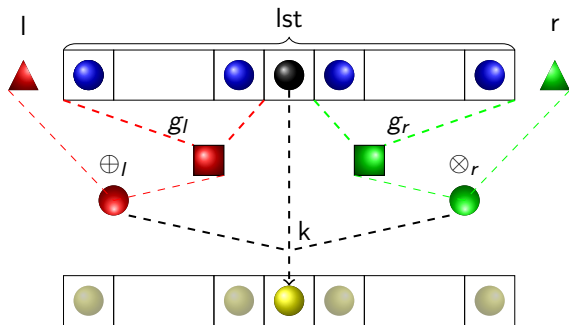
La fonction h est un homomorphisme BSP, ou BH, si elle vérifie les équations suivantes :

$$\left\{ \begin{array}{ll} bh [] \mid r = [] & (BH\text{-nil}) \\ bh [a] \mid r = [k \ a \mid r] & (BH\text{-Singleton}) \\ bh (x ++ y) \mid r = bh \ x \mid (g_r \ y \otimes_r \ r) ++ \\ \quad bh \ y \mid (\oplus_l \ g_l \ x) \ r & (BH\text{-Concat}) \end{array} \right.$$

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



$$\forall \text{bh } l \text{ } lst \text{ } r \text{ } i, \text{nth } i \text{ (bh } l \text{ } lst \text{ } r) =$$

$$k$$

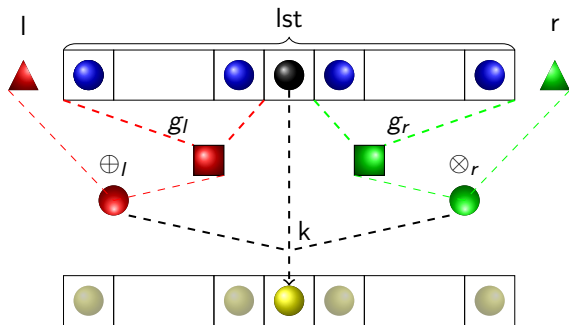
$$(l \oplus_l (g_l (\text{nth } i (\text{inits } lst))))$$

$$(\text{nth } i \text{ } lst)$$

$$(g_r (\text{nth } i (\text{tails } lst))) \otimes_r r$$

L'homomorphisme BSP - Calcul séquentiel

► *bh_comp*



$$\forall \text{ bh } l \text{ lst } r \text{ } i, \text{ nth } i \text{ (bh } l \text{ lst } r) =$$

$$k$$

$$(l \oplus_l (g_l (\text{nth } i (\text{inits } \text{lst}))))$$

$$(\text{nth } i \text{ lst})$$

$$(g_r (\text{nth } i (\text{tails } \text{lst}))) \otimes_r r$$

L'homomorphisme BSP - Calcul séquentiel

► *bh_comp*

==

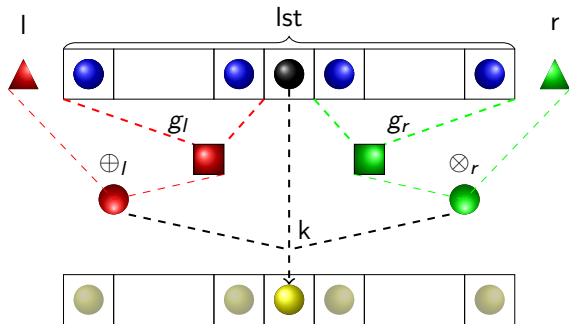
► *bh_seq*

map3 *k*

(*map* *g_l* (*inits* *l*))

l

(*map* *g_r* (*tails* *l*))



$\forall \text{ bh } l \text{ lst } r \ i, \text{nth } i \text{ (bh } l \text{ lst } r) =$

k

(*l* $\oplus_l$ (*g_l* (*nth* *i* (*inits* *lst*))))

(*nth* *i* *lst*)

(*g_r* (*nth* *i* (*tails* *lst*))) $\otimes_r$ *r*)

L'homomorphisme BSP - Calcul séquentiel

► *bh_comp*

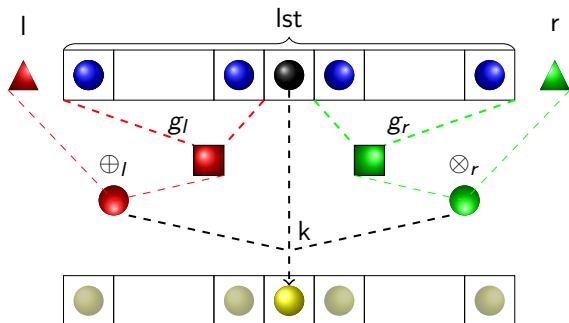
==

► *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*)



$$\forall \text{ bh } l \text{ lst } r \ i, \text{ nth } i \text{ (bh } l \text{ lst } r) =$$

$$k$$

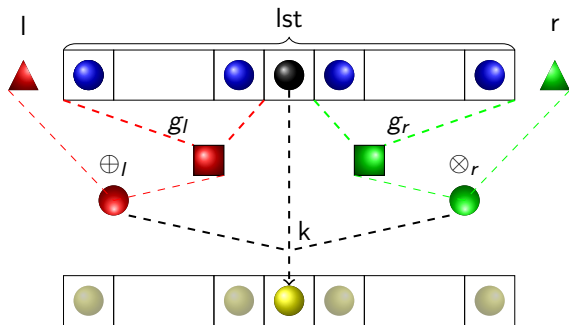
$$(l \oplus_l (g_l (\text{nth } i (\text{inits } \text{lst}))))$$

$$(\text{nth } i \text{ lst})$$

$$(g_r (\text{nth } i (\text{tails } \text{lst}))) \otimes_r r$$

L'homomorphisme BSP - Calcul séquentiel

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



$$\forall \text{ bh } l \text{ lst } r \text{ i, } nth \text{ i } (\text{bh } l \text{ lst } r) =$$

$$k$$

$$(l \oplus_l (g_l (nth \text{ i } (inits \text{ lst}))))$$

$$(nth \text{ i } \text{ lst})$$

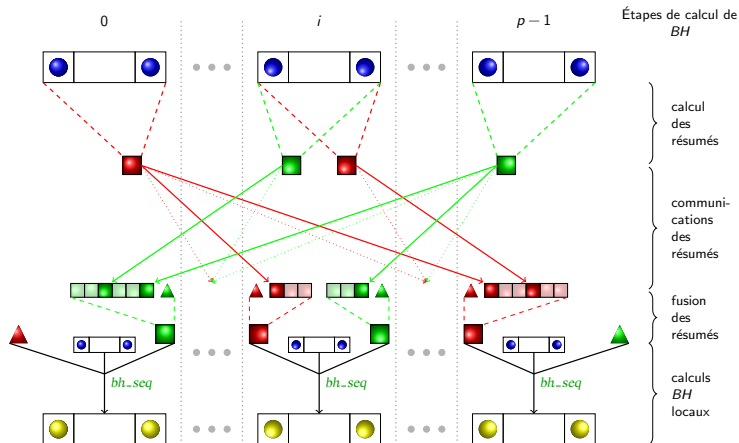
$$(g_r (nth \text{ i } (tails \text{ lst}))) \otimes_r r$$

L'homomorphisme BSP - Modélisation

```
Class BH_PROP ( $f : \text{list } A \rightarrow L \rightarrow R \rightarrow \text{list } B$ ) :=  
{  
   $bh\_k : L \rightarrow A \rightarrow R \rightarrow B$  ;  
   $bh\_gl : \text{list } A \rightarrow L$  ;  
   $bh\_opl : L \rightarrow L \rightarrow L$  ;  
   $bh\_gr : \text{list } A \rightarrow R$  ;  
   $bh\_opr : R \rightarrow R \rightarrow R$  ;  
   $bh\_nil : \forall l\ r, \ f\ []\ l\ r = []$  ;  
   $bh\_singleton : \forall a\ l\ r, \ f\ [a]\ l\ r = [bh\_k\ l\ a\ r]$  ;  
   $bh\_append : \forall (x\ y : \text{list } A)\ l\ r,$   
     $f\ (x++y)\ l\ r =$   
     $f\ x\ l\ (bh\_opr\ (bh\_gr\ y)\ r) ++ f\ y\ (bh\_opl\ l\ (bh\_gl\ x))\ r$   
   $bh\_left\_homomorphism : \text{Homomorphism}'\ bh\_opl\ bh\_gl$  ;  
   $bh\_right\_homomorphism : \text{Homomorphism}'\ bh\_opr\ bh\_gr$  ;  
}
```

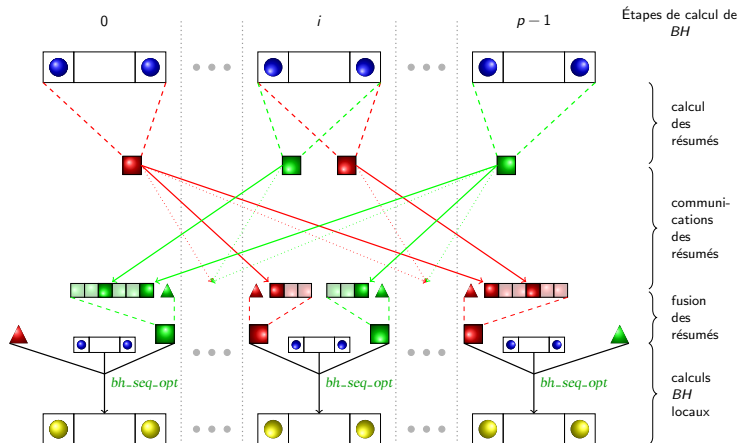

L'homomorphisme BSP - Calcul parallèle

bh_bsm1



L'homomorphisme BSP - Calcul parallèle

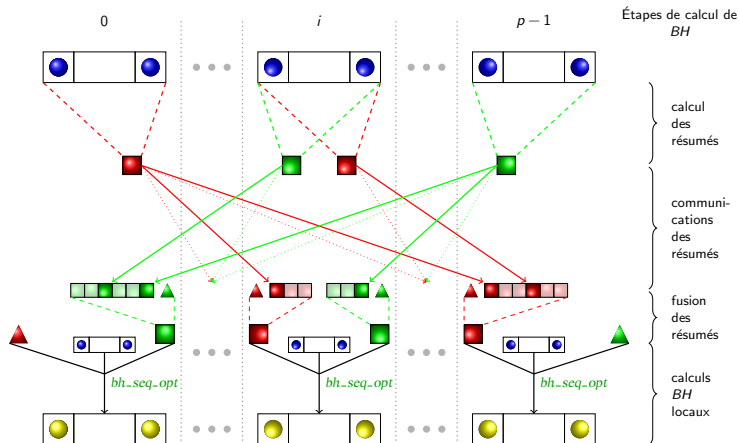
bh_bsm1_opt



L'homomorphisme BSP - Calcul parallèle

bh_bsm1_opt

Parallel (*bh_comp*) (*bh_bsm1_opt*)



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

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{heat } (x \mathbin{++} y) \mid r = \text{heat } x \mid (\text{hd } y \mid r) \mathbin{++} \\ \qquad \qquad \qquad \text{heat } y \mid (\text{last } x \mid r) \end{array} \right.$$

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd } y \ r) \ ++ \\ \qquad \qquad \qquad \text{heat } y \ (\text{last } x \mid) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \ \otimes_r \ r) \ ++ \\ \qquad \qquad \qquad \text{bh } y \ (\mid \oplus_l \ g_l \ x) \ r \end{array} \right.$$

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd } y \ r) \ ++ \\ \qquad \qquad \qquad \text{heat } y \ (\text{last } x \mid) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \ \otimes_r \ r) \ ++ \\ \qquad \qquad \qquad \text{bh } y \ (l \oplus_l \ g_l \ x) \ r \end{array} \right.$$

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd_option } y \ll r) \ ++ \\ \quad \text{heat } y \mid (\text{last_option } x \gg l) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \otimes_r r) \ ++ \\ \quad \text{bh } y \mid (l \oplus_l g_l \ x) \ r \end{array} \right.$$

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd_option } y \ll r) \ ++ \\ \quad \text{heat } y \mid (\text{last_option } x \gg l) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \otimes_r r) \ ++ \\ \quad \text{bh } y \mid (l \oplus_l g_l \ x) \ r \end{array} \right.$$

heatSeq *heatSeqBH* BH_PROP BH_PROP_parallelizable

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

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd_option } y \ll r) \ ++ \\ \quad \text{heat } y \ (\text{last_option } x \gg l) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \otimes_r r) \ ++ \\ \quad \text{bh } y \ (l \oplus_l g_l \ x) \ r \end{array} \right.$$

heatSeq *heatSeqBH* BH_PROP BH_PROP_parallelizable

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

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd_option } y \ll r) \ ++ \\ \quad \text{heat } y \ (\text{last_option } x \gg l) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \otimes_r r) \ ++ \\ \quad \text{bh } y \ (l \oplus_l g_l \ x) \ r \end{array} \right.$$

heatSeq *heatSeqBH* BH_PROP BH_PROP_parallelizable

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

Propriété de heat

$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd_option } y \ll r) \ ++ \\ \quad \text{heat } y \ (\text{last_option } x \gg l) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \otimes_r r) \ ++ \\ \quad \text{bh } y \ (l \oplus_l g_l \ x) \ r \end{array} \right.$$

heatSeq *heatSeqBH* BH_PROP BH_PROP_parallelizable

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

Propriété de heat

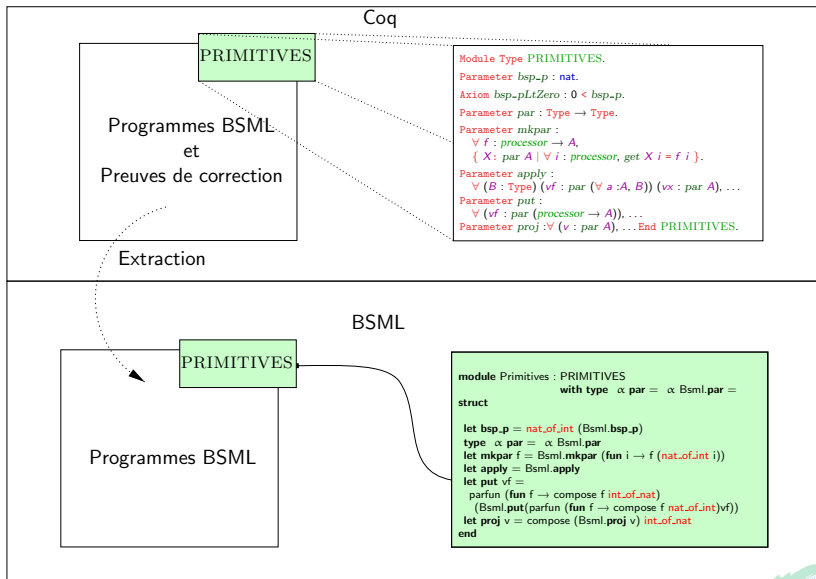
$$\left\{ \begin{array}{l} \text{heat } [] \mid r = [] \\ \text{bh } [] \mid r = [] \\ \text{heat } [a] \mid r = [\text{Formula } a \mid r] \\ \text{bh } [a] \mid r = [k \ a \mid r] \\ \text{heat } (x \ ++ \ y) \mid r = \text{heat } x \mid (\text{hd_option } y \ll r) \ ++ \\ \qquad \qquad \qquad \text{heat } y \ (\text{last_option } x \gg l) \ r \\ \text{bh } (x \ ++ \ y) \mid r = \text{bh } x \mid (g_r \ y \otimes_r r) \ ++ \\ \qquad \qquad \qquad \text{bh } y \ (l \oplus_l g_l \ x) \ r \end{array} \right.$$

heatSeq *heatSeqBH* BH_PROP BH_PROP_parallelizable

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

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

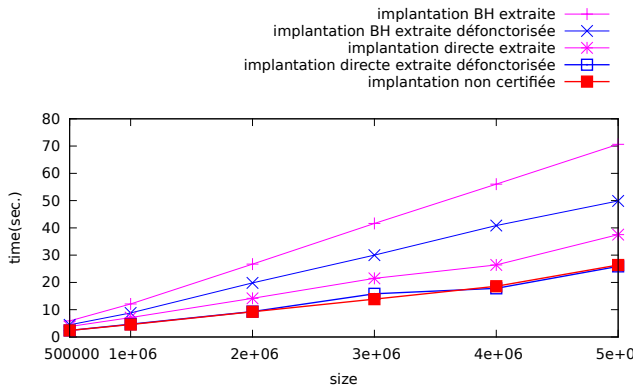
Extraction



Expérimentation



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



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

		Spécification	Preuve
Modélisation de BSMML	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 **vérifiées**



Exécutions parallèles



Merci

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

▸ 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  $p \Rightarrow$  (fun  $i \Rightarrow i+1$ )))  
(mkpar (fun  $p \Rightarrow proj1\_sig\ p$ )).
```


Exemple - Évaluation symbolique BSML

Goal

$\forall p : \text{processor},$
 $\text{get } Sp\ p = 'p + 1.$

Proof.

Exemple - Évaluation symbolique BSML

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

```
 $get$   
  ( $proj1\_sig$   
    (apply  
      ( $proj1\_sig$  ( $mkpar$  ( $fun$  ( $_ : processor$ ) ( $i : nat$ )  $\Rightarrow i + 1$ ))))  
      ( $proj1\_sig$  ( $mkpar$  ( $fun$   $p0 : processor \Rightarrow 'p0$ ))))))  $p$   
=  
' $p + 1$ 
```

Exemple - Évaluation symbolique BSML

```
rewrite (fun V1 V2  $\Rightarrow$  proj2_sig (apply V1 V2)).
```

```
(get (proj1_sig (mkpar (fun (- : processor) (i : nat)  $\Rightarrow$  i + 1))) p)  
(get (proj1_sig (mkpar (fun p0 : processor  $\Rightarrow$  'p0))) p)  
=  
'p + 1
```

Exemple - Évaluation symbolique BSML

```
rewrite (fun f  $\Rightarrow$  proj2_sig (mkpar f)).
```

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

Exemple - Évaluation symbolique BSML

```
rewrite (fun f  $\Rightarrow$  proj2_sig (mkpar f)).
```

```
(fun (i : nat)  $\Rightarrow$  i + 1)  
'p  
=  
'p + 1
```

Exemple - Évaluation symbolique BSML

```
reflexivity.
```

```
Qed.
```

Unnamed_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 :  $\forall i$ , get v i  $\neq$  nil) :

  { vr : par A |  $\forall (i$  :processor),
    get vr i =
      if ( i == firstProc ) then l else sLast (get v (i-1))
  }  $\times$ 
  { vr : par A |  $\forall (i$  :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 :  $\forall i$ , get v i  $\neq$  nil) :
  { vr : par (option A) |  $\forall (i : \text{processor})$ ,
    get vr i = Some ( if ( i == firstProc ) then l else
sLast (get v (i-1)) ) }  $\times$ 
  { vr : par (option A) |  $\forall (i : \text{processor})$ ,
    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)  $\Rightarrow$ 
  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  $\Rightarrow$  pid-1)) _ _ ) ,
```

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod bsp\_p)$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == (i + dec) mod bsp_p) ) then Some / else None
  ))
  (v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod bsp\_p)$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == ((i + dec) mod bsp_p) ) then Some / else None
  ))
  v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod \text{bsp\_p})$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == ((i + dec) mod bsp_p) ) then Some / else None
  ))
  v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod \text{bsp\_p})$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == (i + dec) mod bsp_p) then Some / else None
  ))
  v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod \text{bsp\_p})$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == (i + dec) mod bsp_p) then Some / else None
  ))
  (v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod \text{bsp\_p})$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == (i + dec) mod bsp_p) then Some / else None
  ))
  (v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod bsp\_p)$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == ((i + dec) mod bsp_p) ) then Some / else None
  ))
  v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...


```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod \text{bsp\_p})$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == ((i + dec) mod bsp_p) ) then Some / else None
  ))
  v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

```

Next Obligation. ...

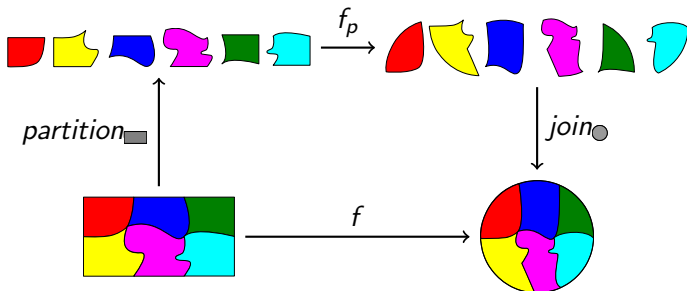
```

Program Definition shift A dec (v : par A) :
  { vr : par A |  $\forall i, \text{get } v \ i = \text{get } vr \ ((i+dec) \bmod bsp\_p)$  } :=
let received := put
  (apply (mkpar (fun (i : processor) / (j : processor)  $\Rightarrow$ 
    if (j == ((i + dec) mod bsp_p) ) then Some / else None
  ))
  v)
  in
  parfun (@noSome _)
    (parSig
      (apply
        (mkpar (fun (i : processor) (f : processor  $\rightarrow$  _)  $\Rightarrow$ 
          f ((bsp_p - (dec mod bsp_p) + i) mod bsp_p)))
        received)
      (fun a  $\Rightarrow$  a  $\neq$  None) _ ) .

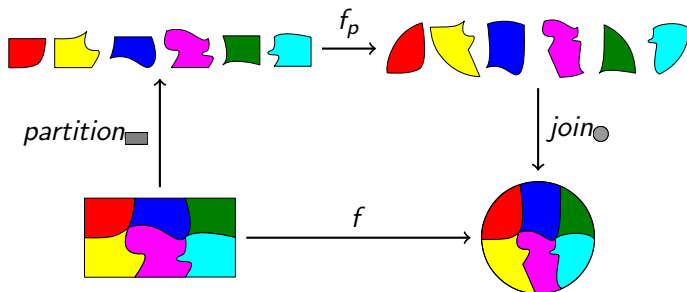
```

Next Obligation. ...

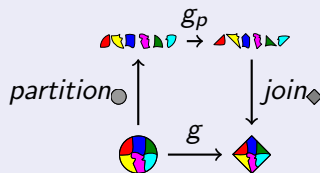
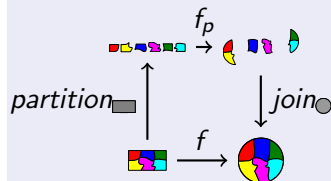
Parallélisation correcte



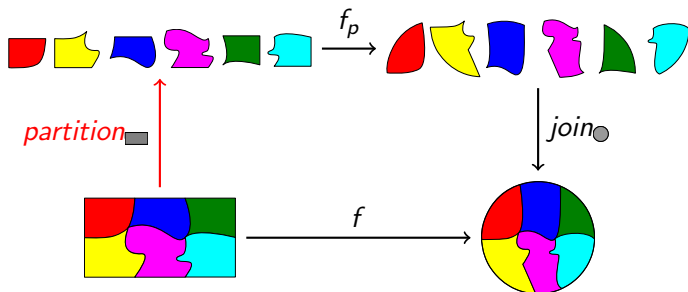
Parallélisation correcte



Composabilité : $g_p \circ f_p$?



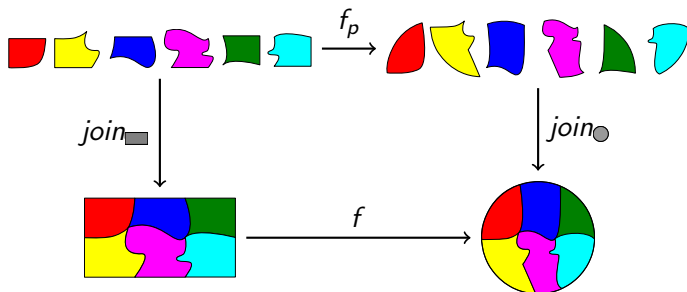
Parallélisation correcte



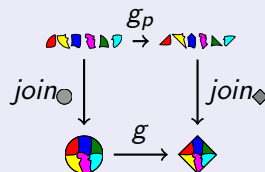
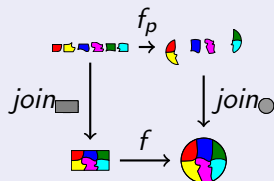
Composabilité : $g_p \circ f_p$?



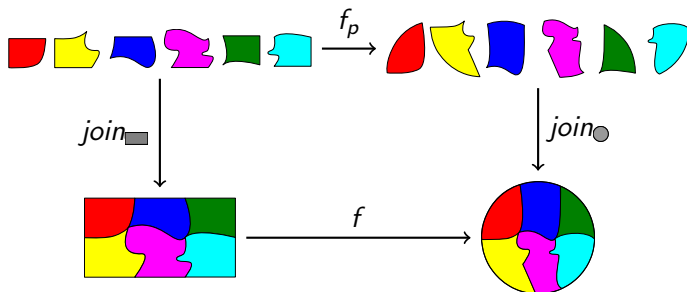
Parallélisation correcte



Composabilité : $g_p \circ f_p$?



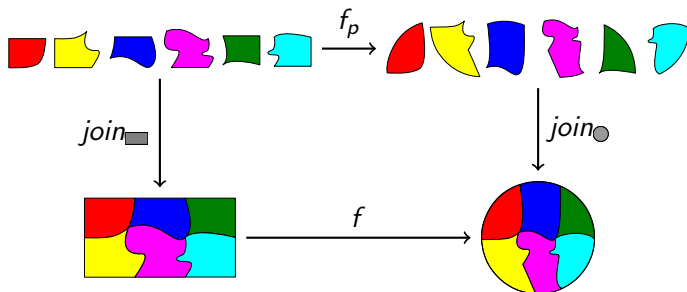
Parallélisation correcte



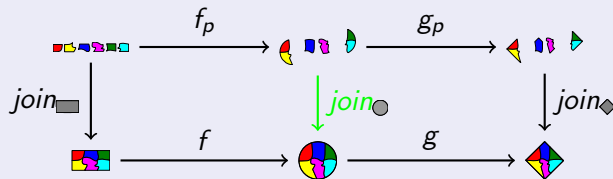
Composabilité : $g_p \circ f_p$?



Parallélisation correcte



Parallélisation correcte composable $g_p \circ f_p$



Fin

