Constructive algorithmic in Coq

Julien Tesson

Kochi University of Technology

FraDeCoPP,
15 mai 2012

Parallel programing

Structured Parallelism

- Algorithmic skeletons
- Bulk Synchronous Parallelism
- Fonctional Parallele Programing

Algorithmic Skeletons

Roots

- Cole 1989
- Patterns of parallel programs

Data Parallelism

- Distributed data structures
- Collectives operations (map)
- Reduction (scan)
- Distribution / balancing operations

Pros

Separation of programing model and execution model

- *Easy* to use
- Optimised implementation for each targeted architecture

Cons

- Limited number of data structures and operations

Bulk Synchronous Parallelism (BSP)

Roots

Valiant & McColl, années 90

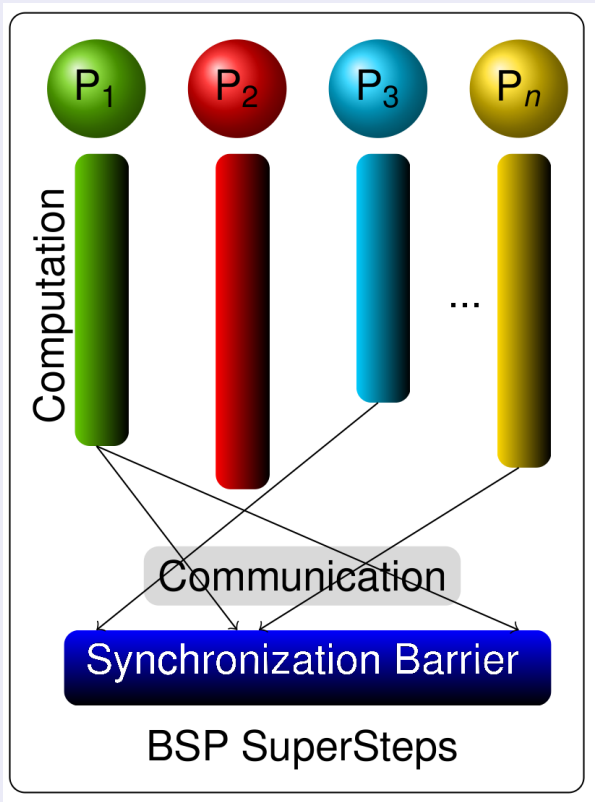
three models

- Abstract architecture
- Execution model
- Cost model

BSP computer

- p memory/processor couples *(of speed r)*
- One communication network *(of speed g)*
- One synchronisation unit *(in time L)*

Execution model



Cost model

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

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

Bulk Synchronous Parallel ML

Conception principles

- A reduced number of parallel primitives
- Universal for bulk synchronous parallelism
- Global view of the parallel program
- A simple formal semantics

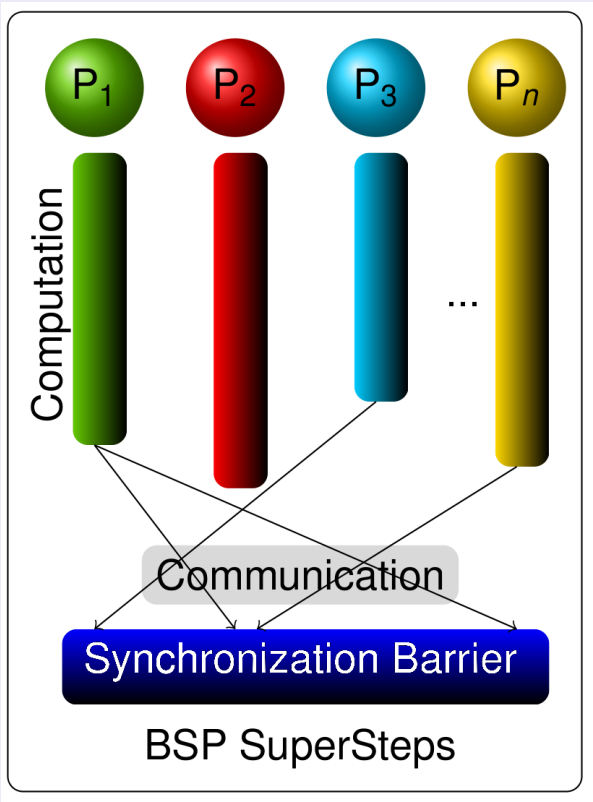
BSML

A functional language

\+ a parallel data structure (without nesting)

\+ parallel operations on this structure

Modèle d'exécution BSP



Tools

- The Coq proof assistant
- The BSML functional parallel language

Overview

BSML Primitives

Algorithmic skeletons

Programs and correctness proofs

Extraction

Extracted BSML Programs

BSML primitives implantation

BSML Compiling

Parallel executions

Proof of program correctness in a proof assistant

Deep embedding

- Abstract syntax tree

\+ semantic rules

Shallow embedding

- Proof assistant language as programming language

\+ axiomatisation

The Coq proof asssitant

Program

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

Lemma

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

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

The Coq proof asssitant

Strongly specified program + 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.
```

Program

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

The Coq proof asssitant

Strongly specified program + 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.
```

Program

```
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')
```

Coq embedding of BSML

Parallel vector

- a polymorphe abstract type : 'a par
- A fixed size p : each processor as a value of tye 'a
- Vector nesting forbiden
- Informal notation :


```
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.

⋮
```

Parallel vector creation

- **mkpar** : (int → 'a) → 'a par

(**mkpar** f)

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

$\vdots$

```
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 }.
```

$\vdots$

Point to point application

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

$$\left(\textbf{apply}\ \begin{array}{|c|c|c|c|}\hline f_0 & f_1 & \cdots & f_{p-1}\\\hline\end{array}\right.$$

$$\left.\begin{array}{|c|c|c|c|}\hline v_0 & v_1 & \cdots & v_{p-1}\\\hline\end{array}\right)$$

$$= \begin{array}{|c|c|c|c|}\hline (f_0\ v_0) & (f_1\ v_1) & \cdots & (f_{p-1}\ v_{p-1})\\\hline\end{array}$$

⋮

```
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

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

$$g_i\ j = f_j\ i$$

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

$\Rightarrow$

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

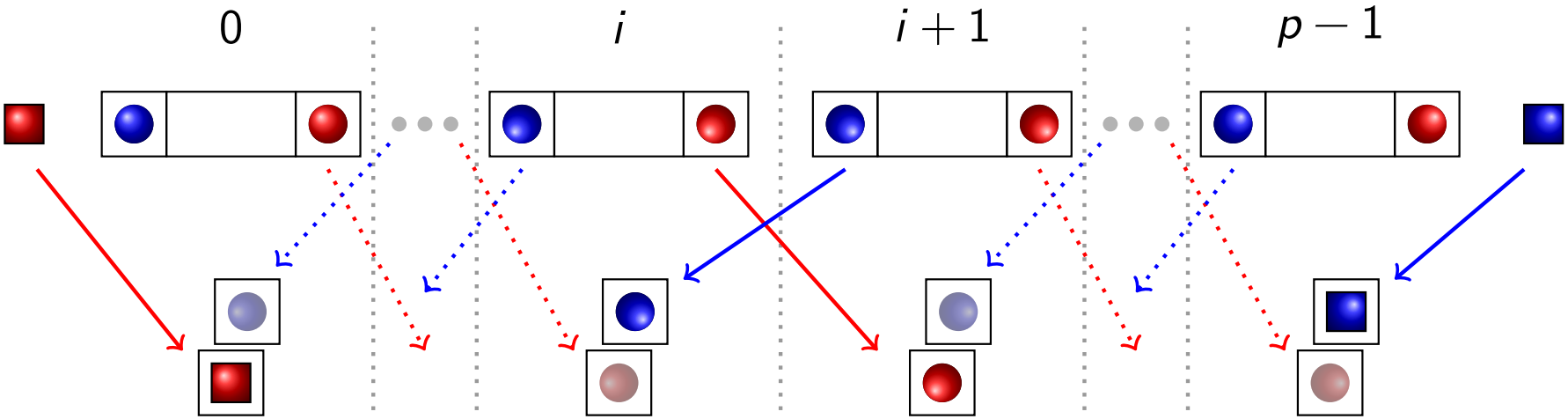
$$\left(\textbf{proj}\ \boxed{v_0 \mid v_1 \mid \cdots \mid v_{p-1}}\right) = \textbf{function}\ \begin{array}{lcl} 0 & \rightarrow & v_0 \\ 1 & \rightarrow & v_1 \\ \vdots & & \\ p-1 & \rightarrow & v_{p-1} \end{array}$$

⋮

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

End PRIMITIVES.
```

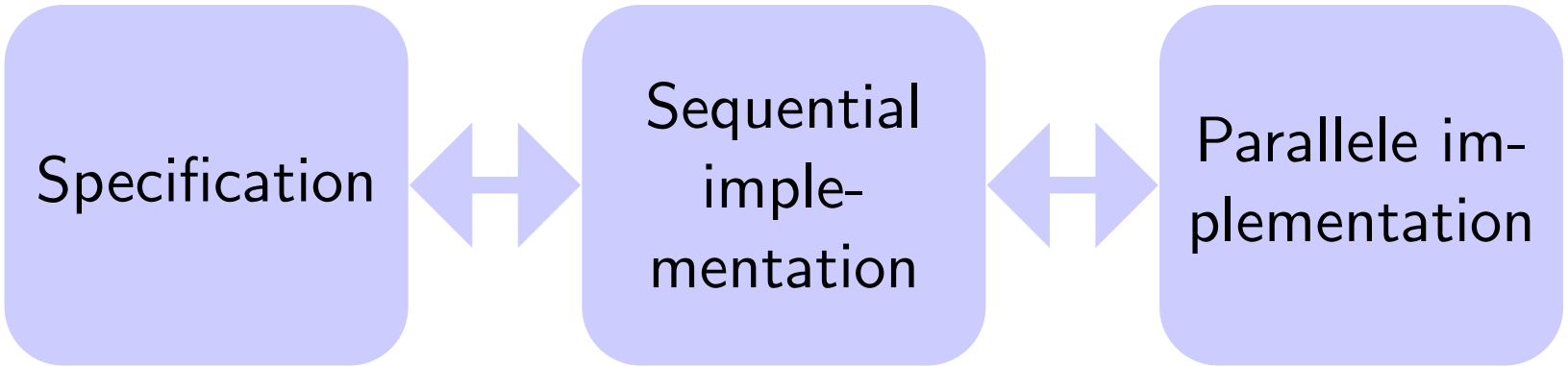
Example - Strong specification of a function

```
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)) }
```



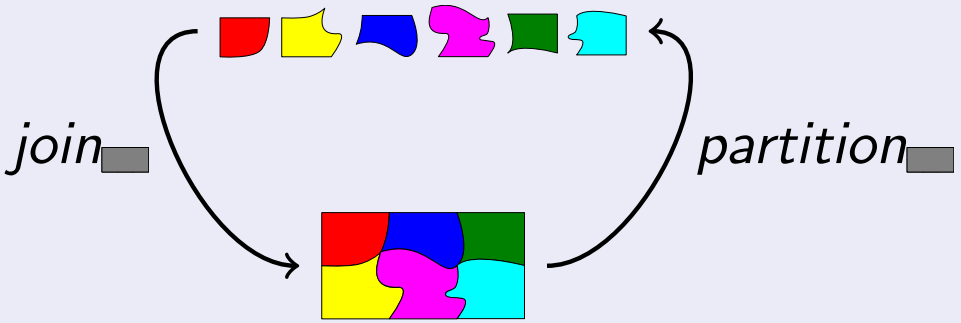
Example - Strong specification of a function

```
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)) }
```



Data distribution

Partitionnable data-structure

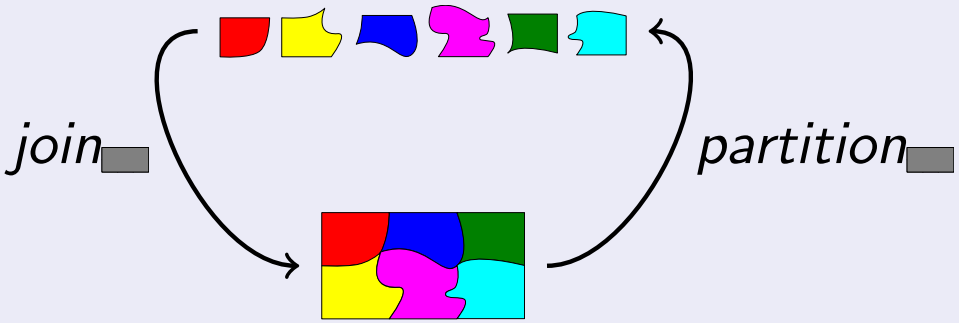


Properties

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

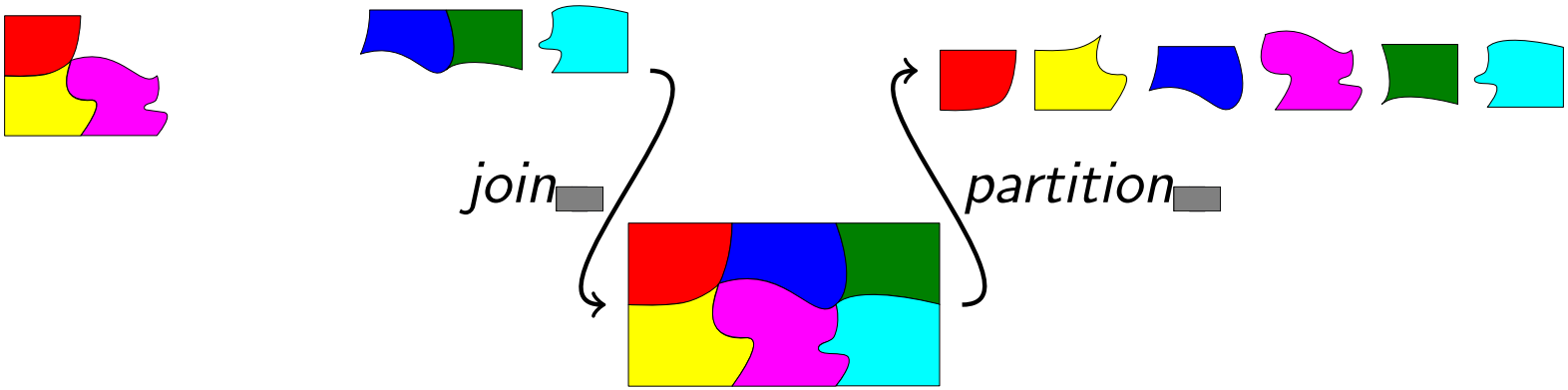
Data distribution

Partitionnable data-structure



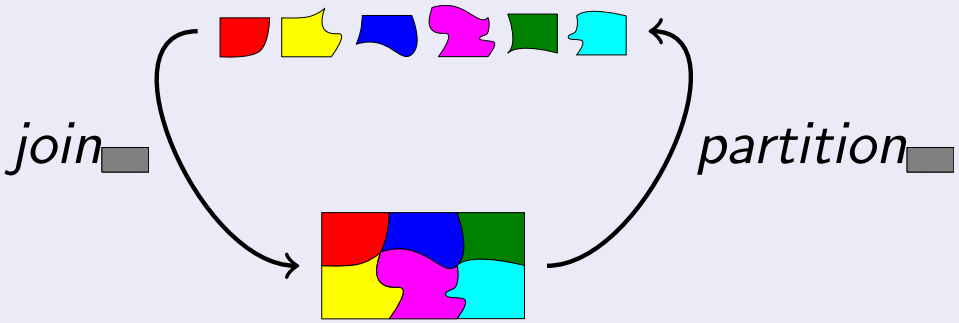
Properties

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



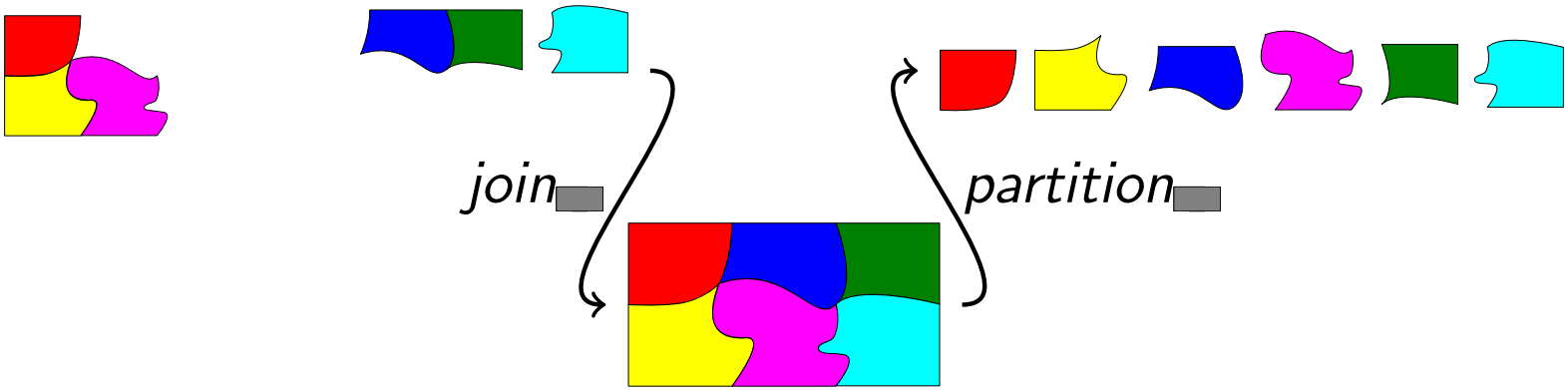
Data distribution

Partitionnable data-structure



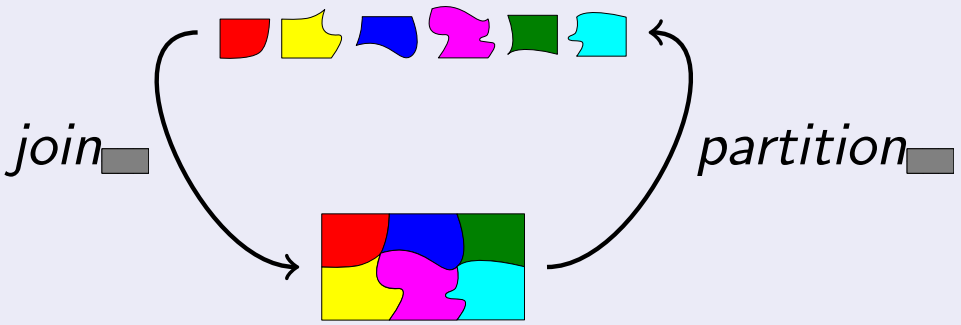
Properties

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



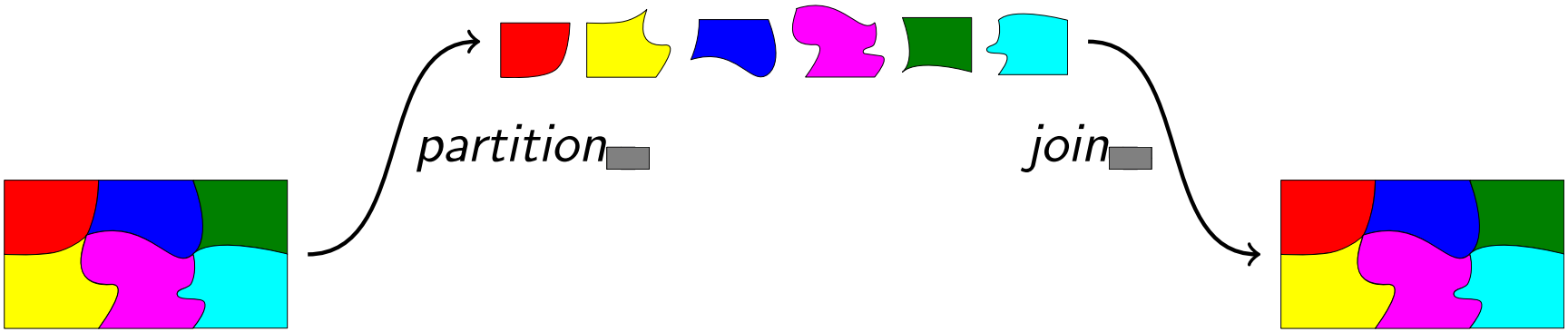
Data distribution

Partitionnable data-structure



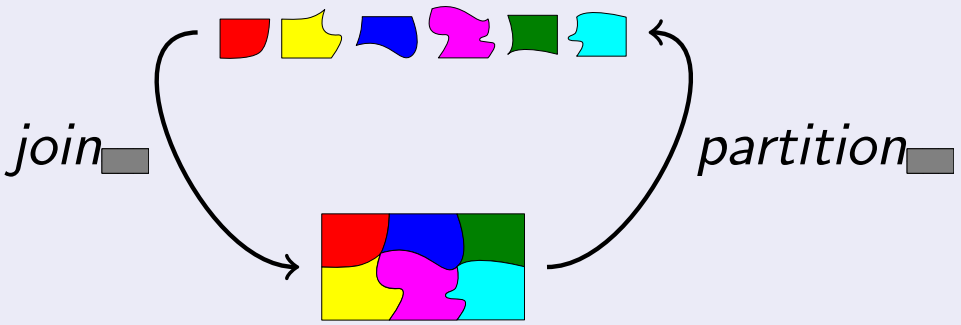
Properties

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



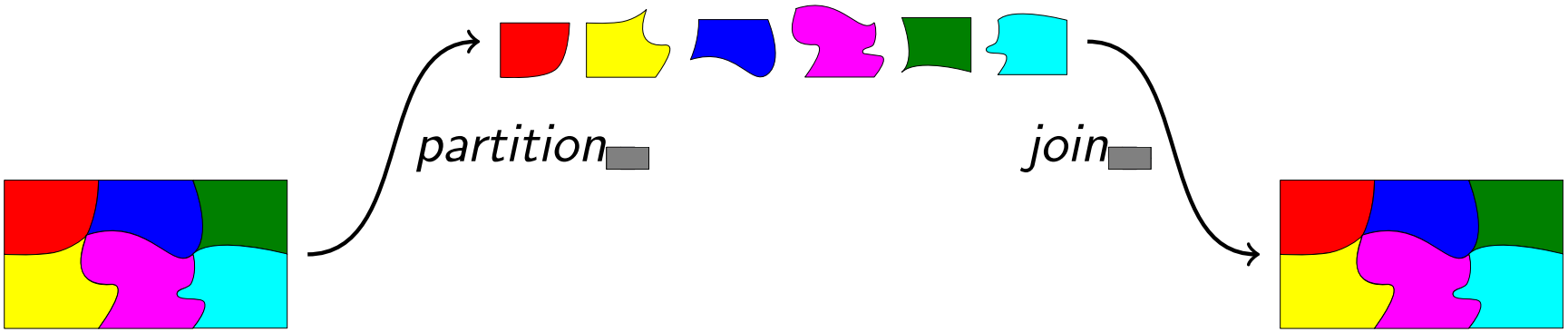
Data distribution

Partitionnable data-structure

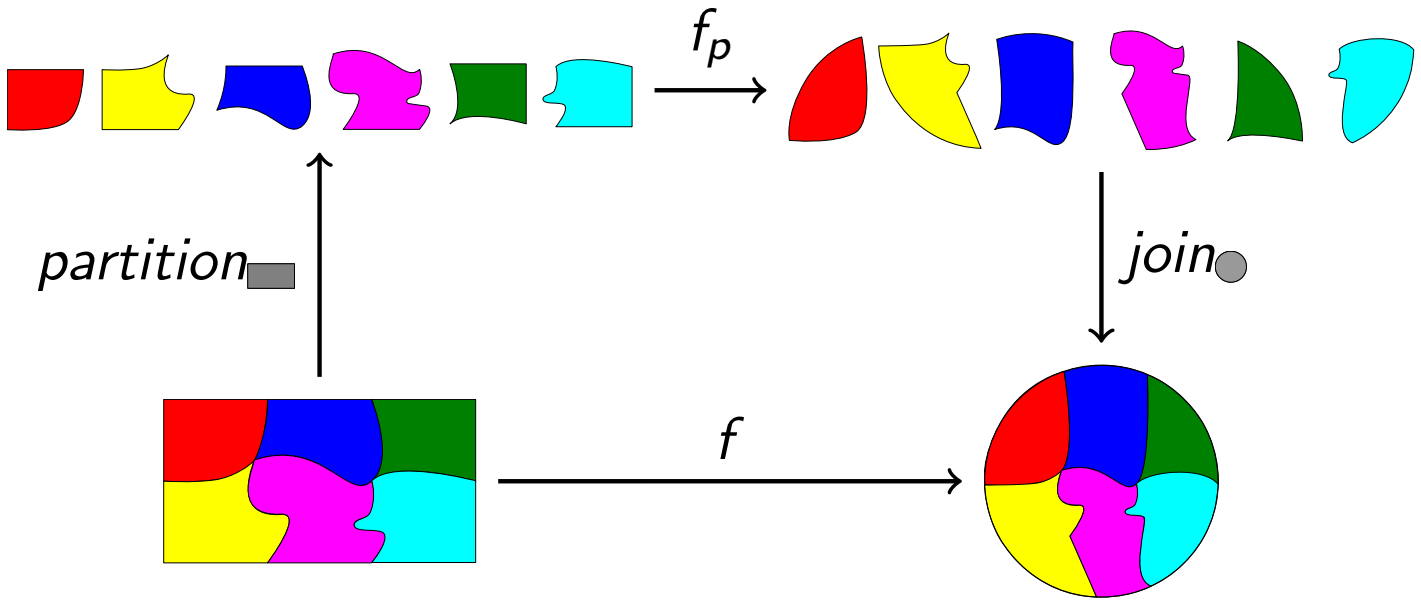


Properties

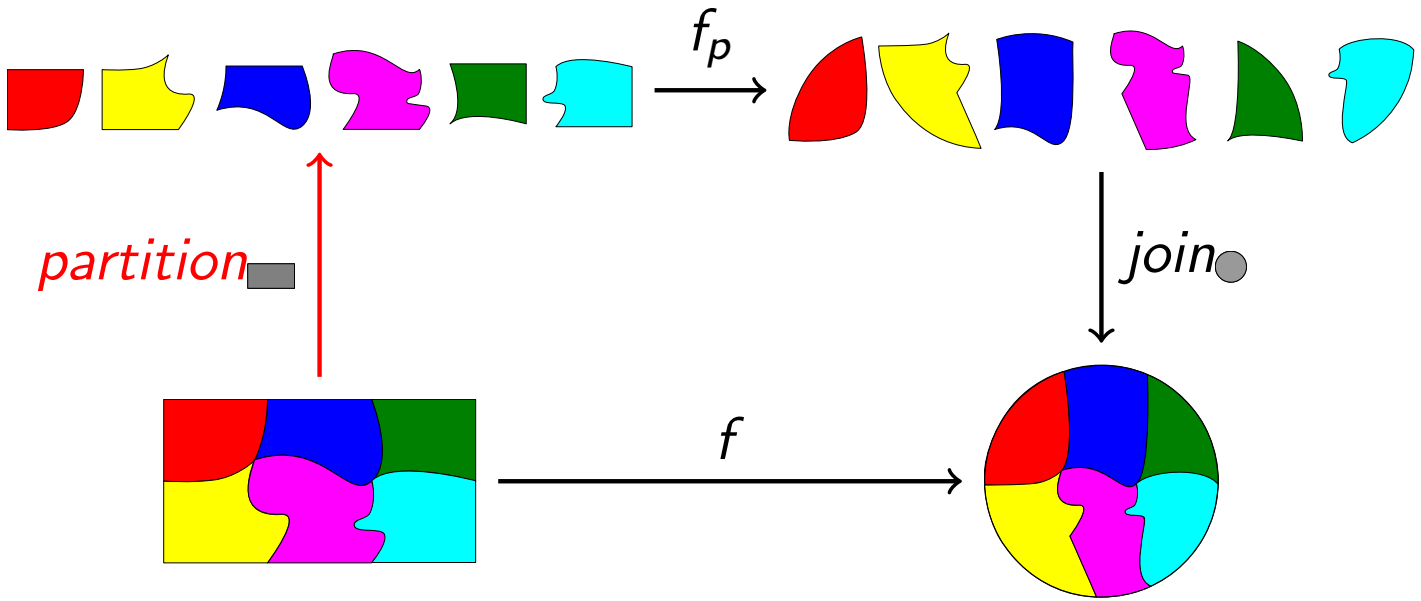
$join_{\blacksquare} \circ partition_{\blacksquare} = id_{\blacksquare}$



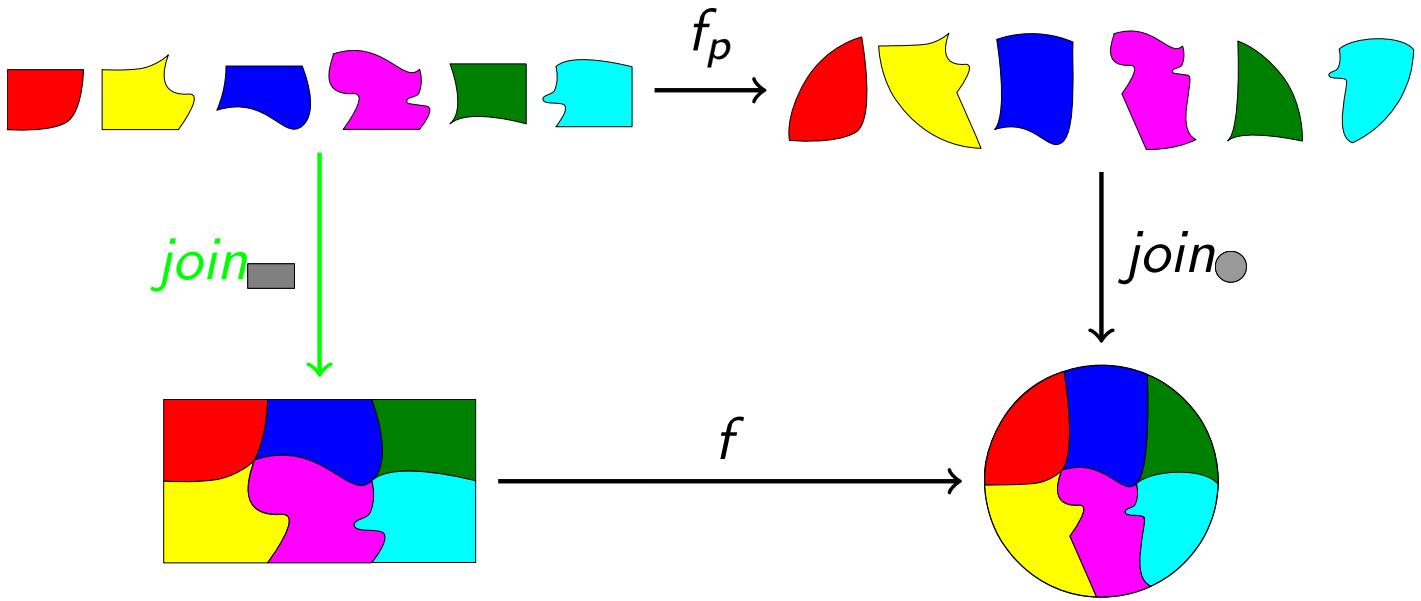
Correct Parallelisation



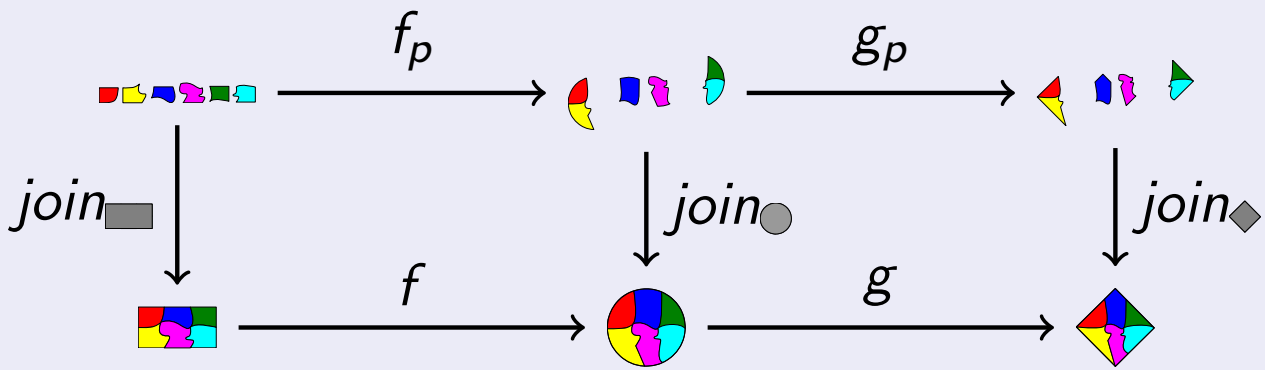
Correct Parallelisation



Correct Parallelisation



Correct and composable parallelisation



Type classes in Coq - overloading

```
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.
```

Type classes in Coq - overloading

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

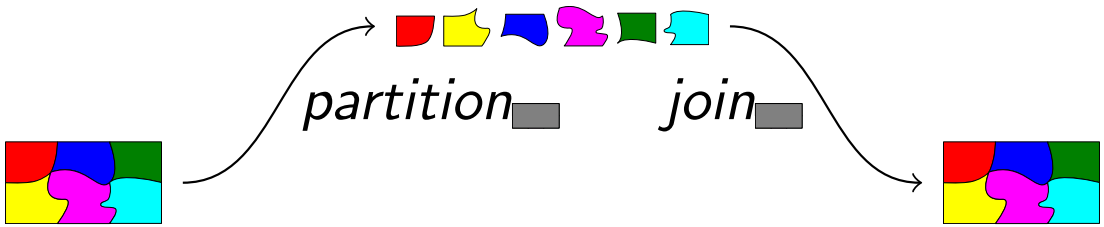
```
Print Instances Associative.
  mult_associatif :
Associative nat mult.

  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.
```

Distribution - Coq Formalisation I

Data-structure distribution

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

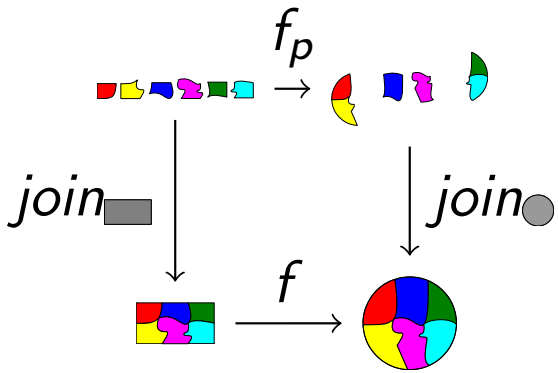


Correct and composable parallelisation - Coq I

Correct parallelisation

```
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)
}.
```

Correct and composable parallelisation - Coq II



Correct and composable parallelisation - Coq III

Instance

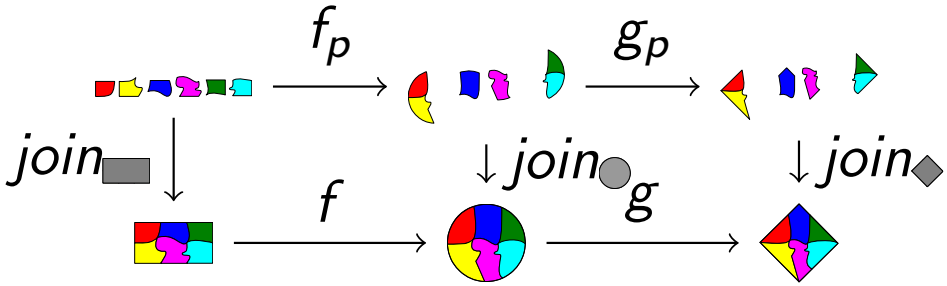
```
Instance FilterParCorrect
(E :Type)(select :E→bool) :Parallel(filter select)(filterPar select).
Proof.
...
```

To declare a parallel implantation, we have to verify the condition *parallel_spec_match*.

Correct and composable parallelisation - Coq IV

Correct Composition

```
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).
```



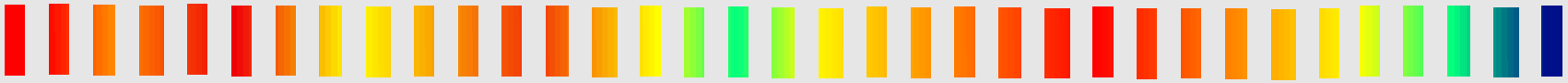
1D Heat diffusion

Diffusion equation :

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

- κ is the diffusion coefficient,
- l and r are constants (bounds temperature, outside the solid)

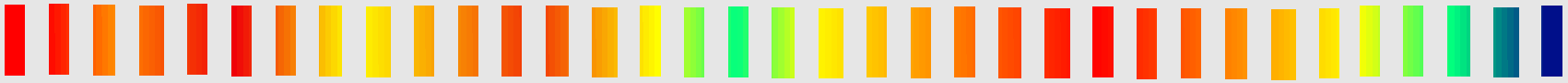
Heat diffusion 1D



A discrete version :

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

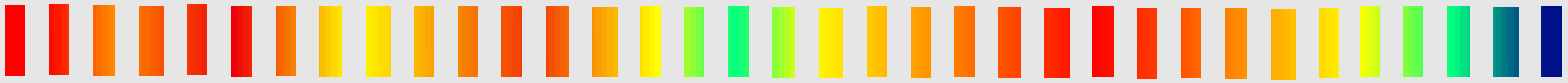
Heat diffusion 1D



A discrete version :

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

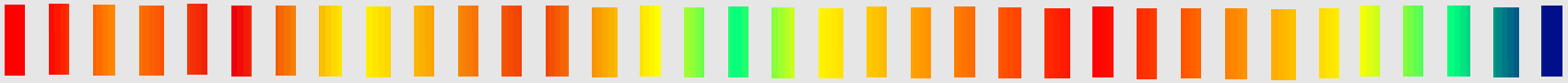
Heat diffusion 1D



Specification :

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

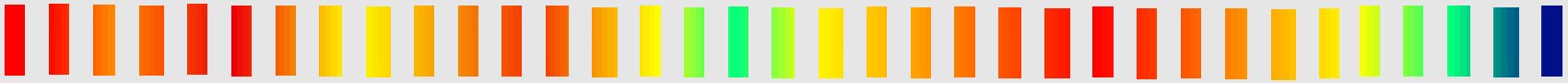
Heat diffusion 1D



Specification :

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

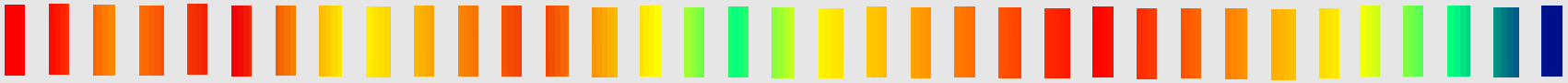
Heat diffusion 1D



Specification :

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

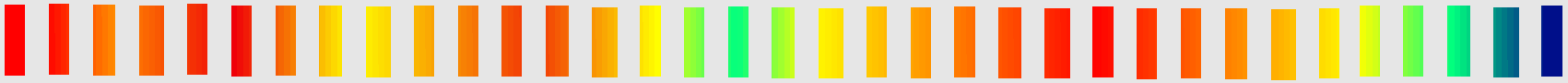
Heat diffusion 1D



Specification :

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

Heat diffusion 1D



Specification :

```
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)
```

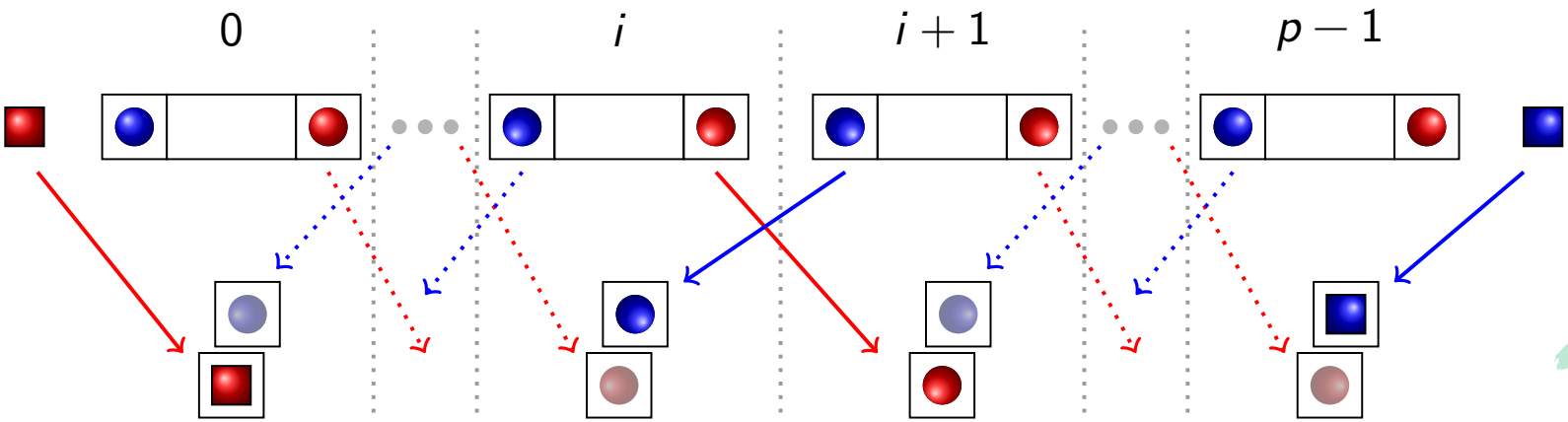
Heat diffusion 1D

Purely functional recursive implementation

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

BSML implementation

```
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.
```



Heat diffusion : correctness

Correctness of sequential implementation

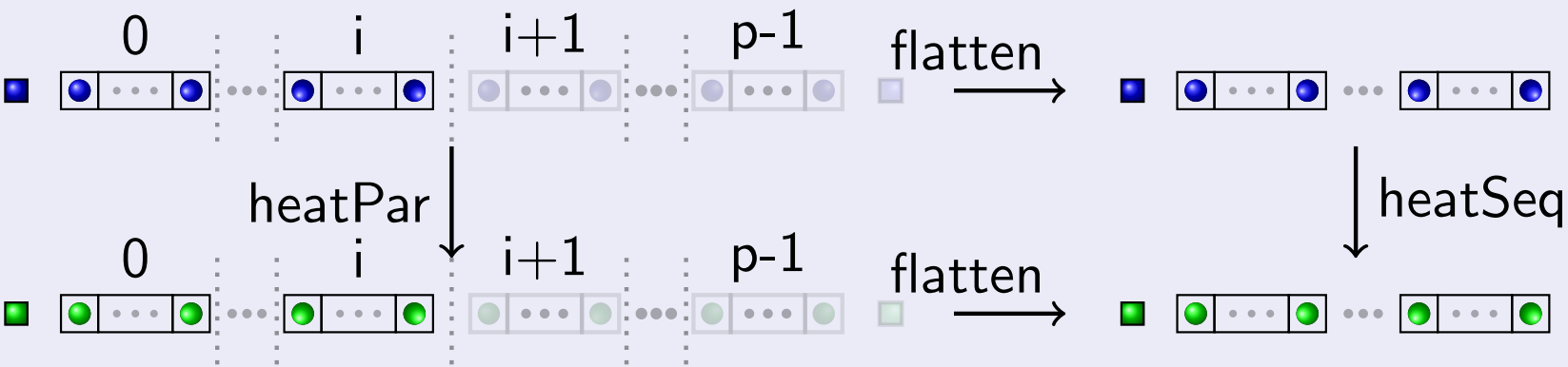
```
Lemma stepSeqFollowsHeatEquationStepSpecification :
    StepSpecification heatSeq.
Proof.
...

  induction u
    ...
    induction i
```

Heat diffusion : correctness - parallel version

Condition for parallelisation

- Parallel version : at least one element per processor
- Sequential version : size of list greater than the number of processors

Sequentialisation lemma

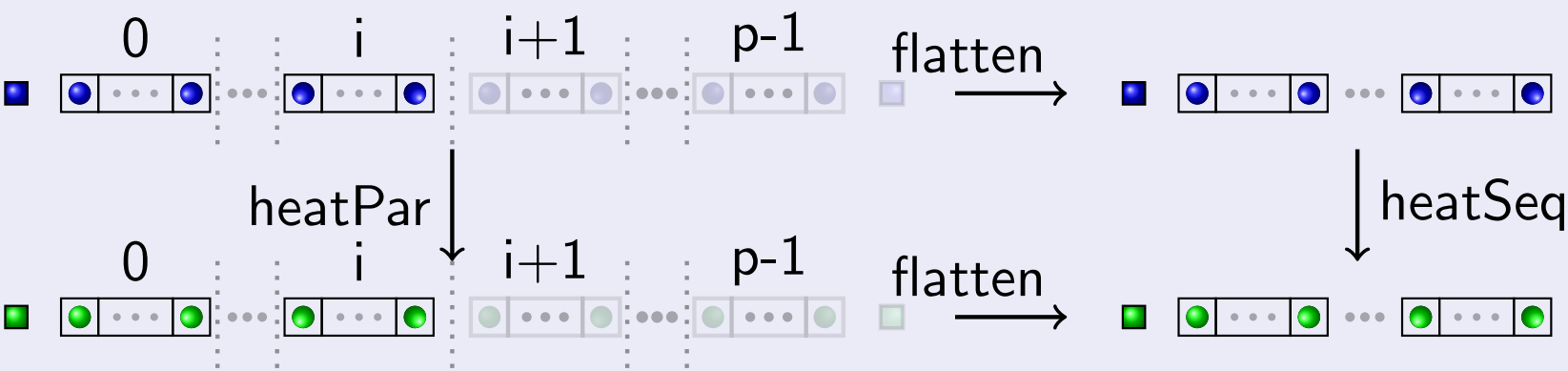


Theorem

The parallel version is a correct composable parallelisation of the sequential implmentation : previous result with $i = p - 1$

Heat diffusion : correctness - parallel version

Sequentialisation lemma



Theorem

The parallel version is a correct composable parallelisation of the sequential implmentation : previous result with $i = p - 1$

Theorem - Coq

Parallel *heatSeq heatPar*

Automatic Parallelisation

Counting elements of size greater than 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).
```

Automatic Parallelisation

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

Automatic Parallelisation

$$\textit{parallel}\ (f\ :=\textit{counting A size})$$

$$\Big|$$

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

Automatic Parallelisation

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

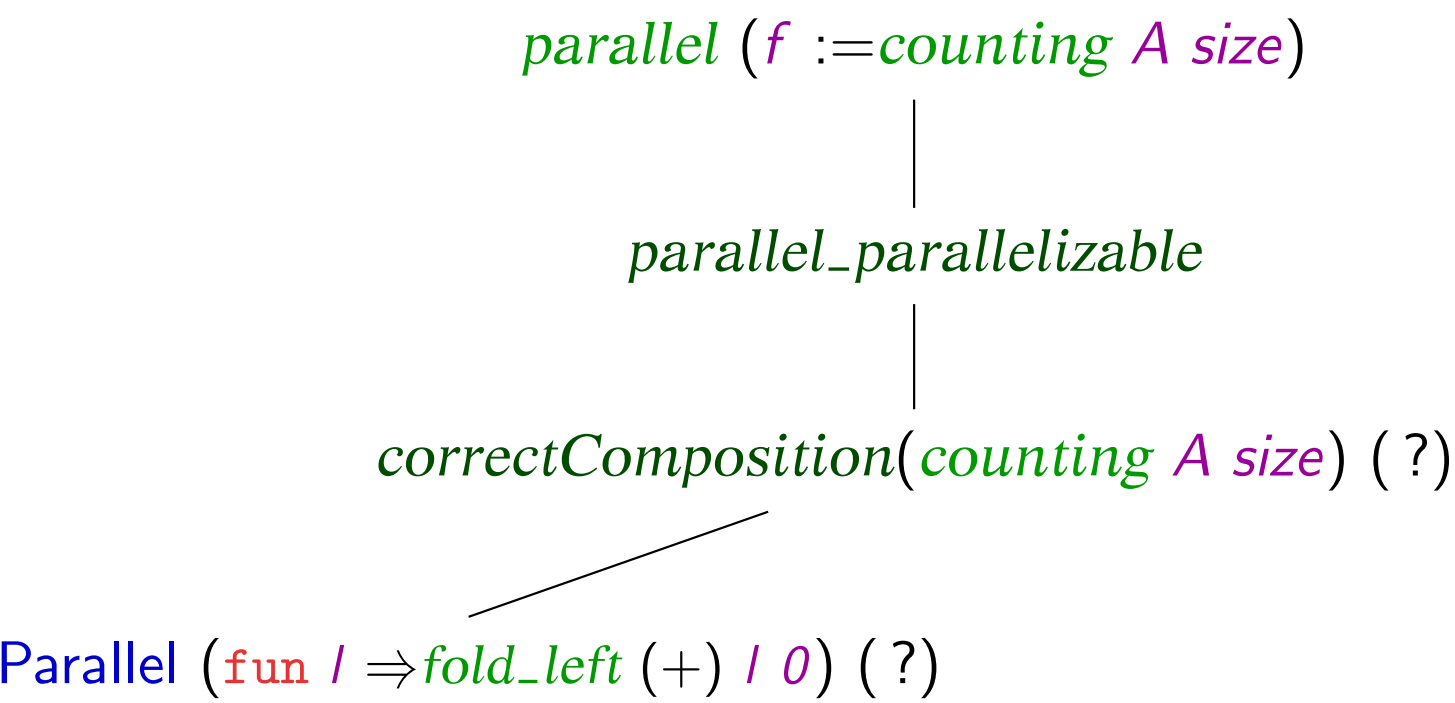
|

parallel_parallelizable

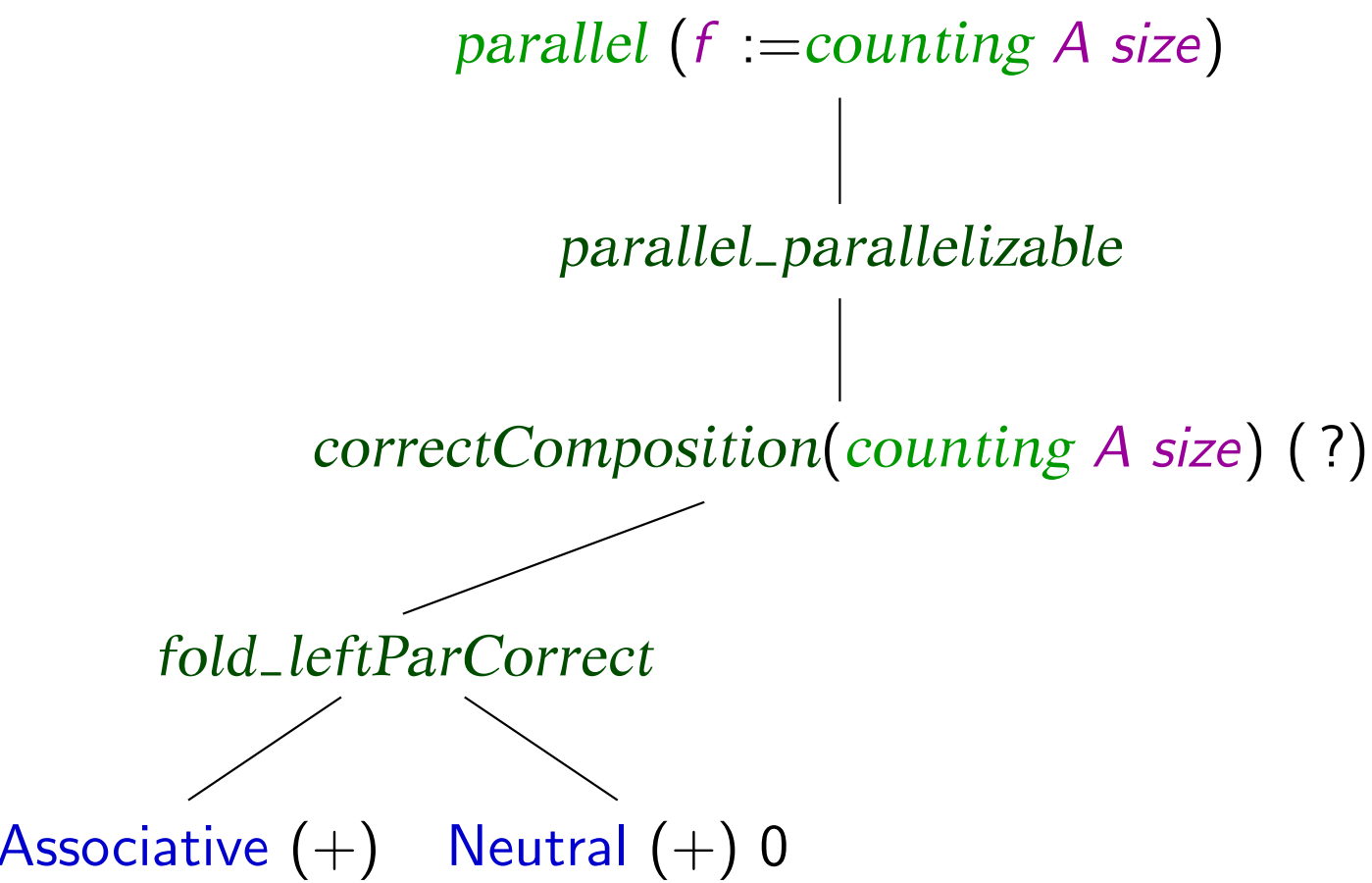
|

Parallel (*counting A size*) (?)

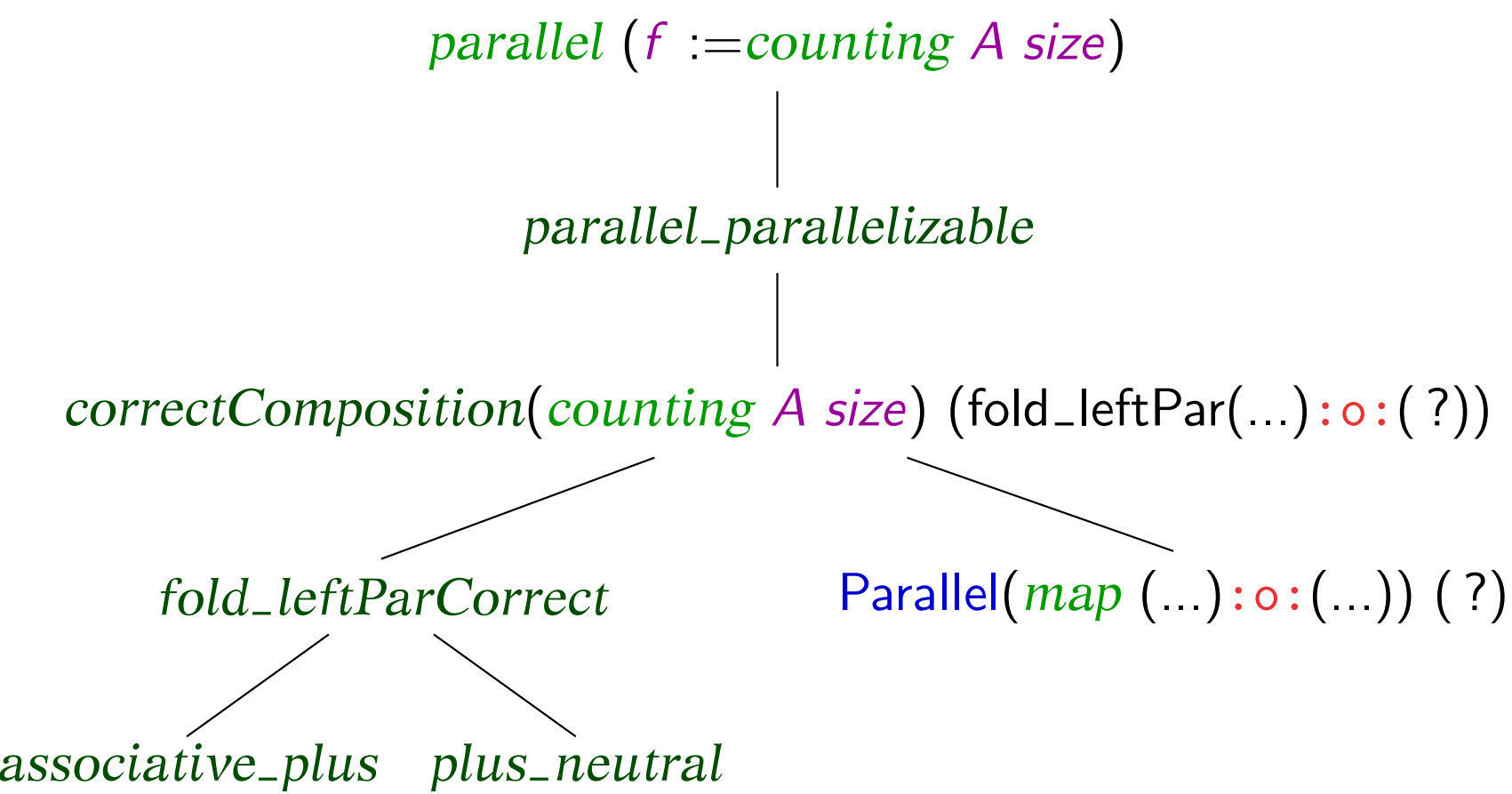
Automatic Parallelisation



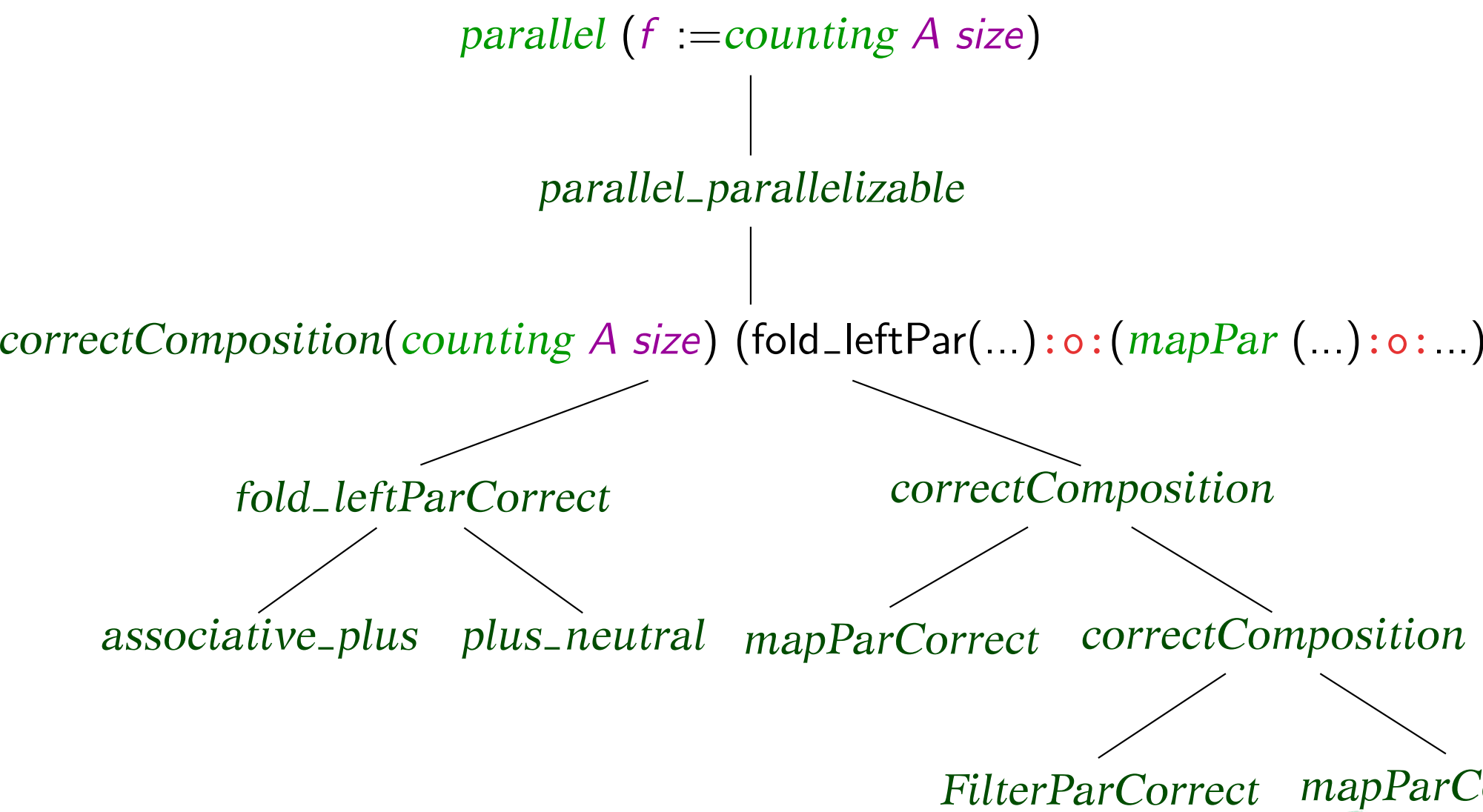
Automatic Parallelisation



Automatic Parallelisation

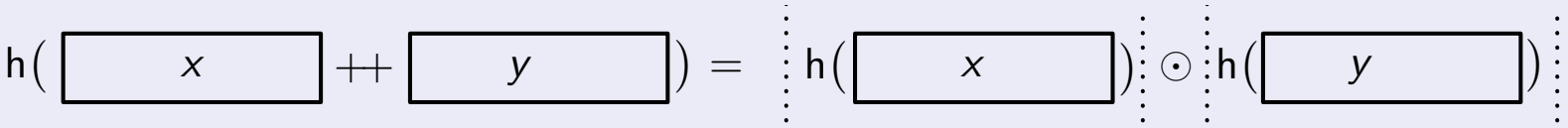


Automatic Parallelisation

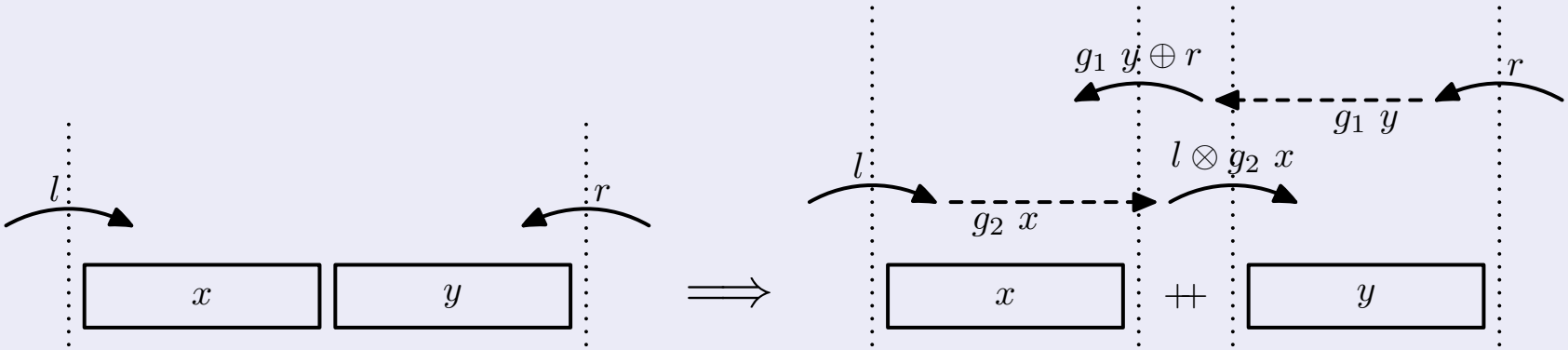


List Homomorphisms and parallelism

List homomorphism



BSP Homomorphism - Hu & Gesbert 2008



BSP Homomorphism : formally

Definition (BH)

a function h is a homomorphism BSP, (or BH), if it satisfy the following equations :

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

Conditions on g_l, g_r, $\oplus_l$ and $\otimes_r$

$(g_l\ [])$ neutral for $\oplus_l$ & $(g_r\ [])$ neutral for $\otimes_r$

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

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

BSP Homomorphism : formally

Definition (BH)

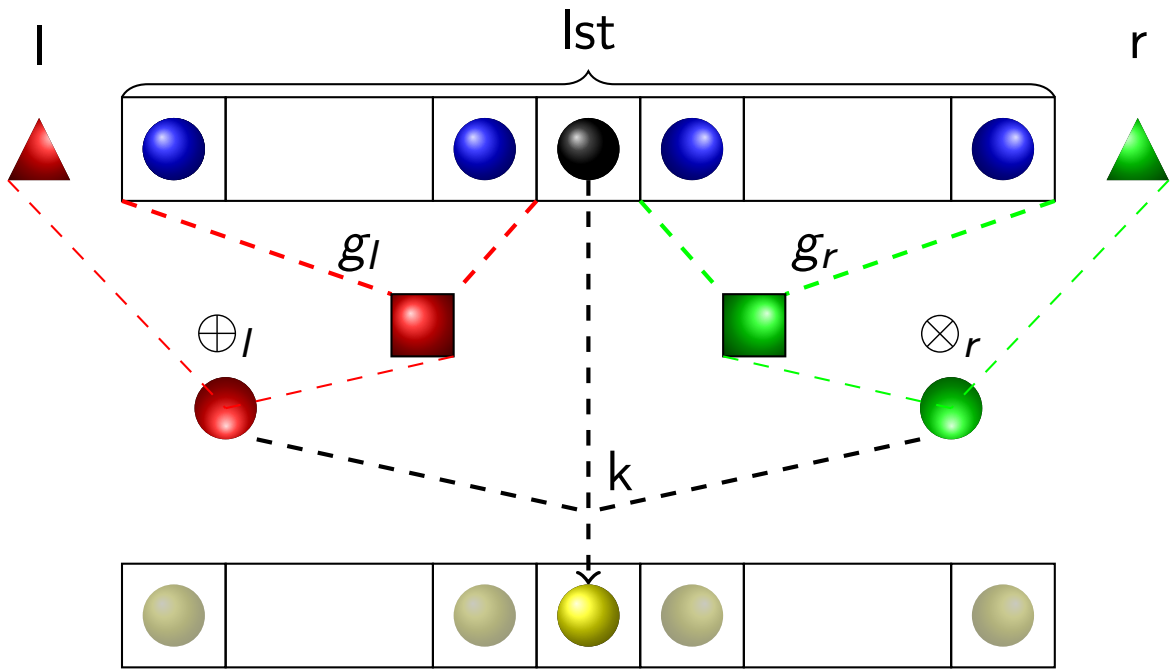
a function h is a homomorphism BSP, (or BH), if it satisfy the following equations :

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

Sufficient conditions on g_l, g_r, $\oplus_l$ and $\otimes_r$

g_l, g_r are homomorphisms associated to operators $\oplus_l$ and $\otimes_r$

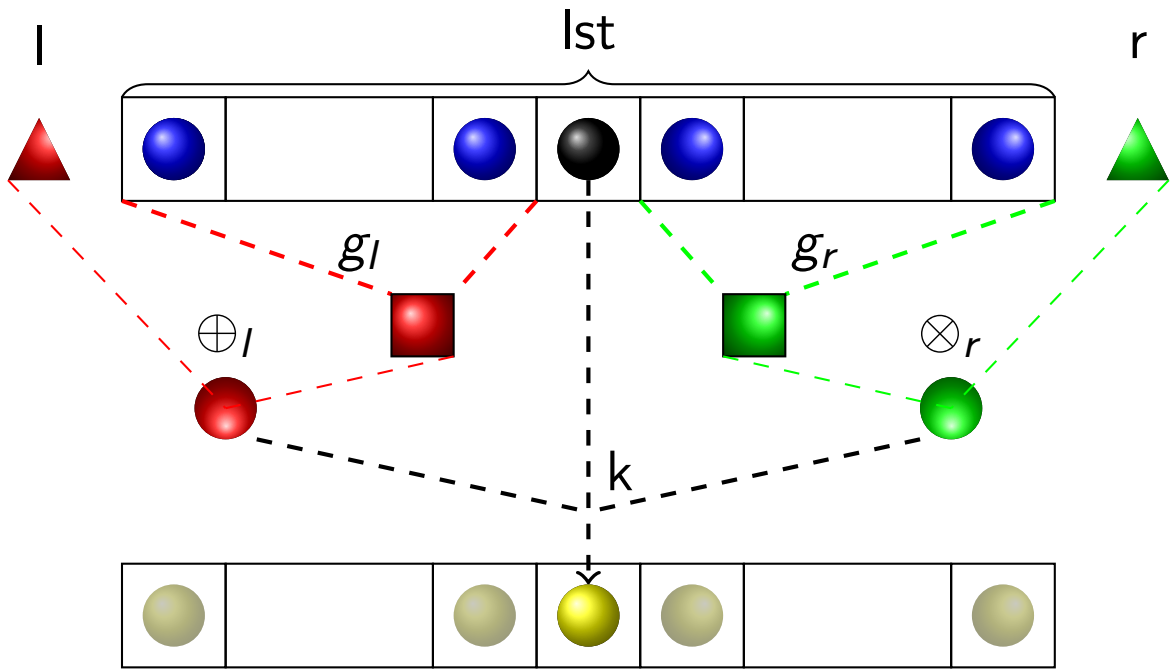
The BSP homomorphism - sequential computation



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

The BSP homomorphism - sequential computation

- *bh_comp*

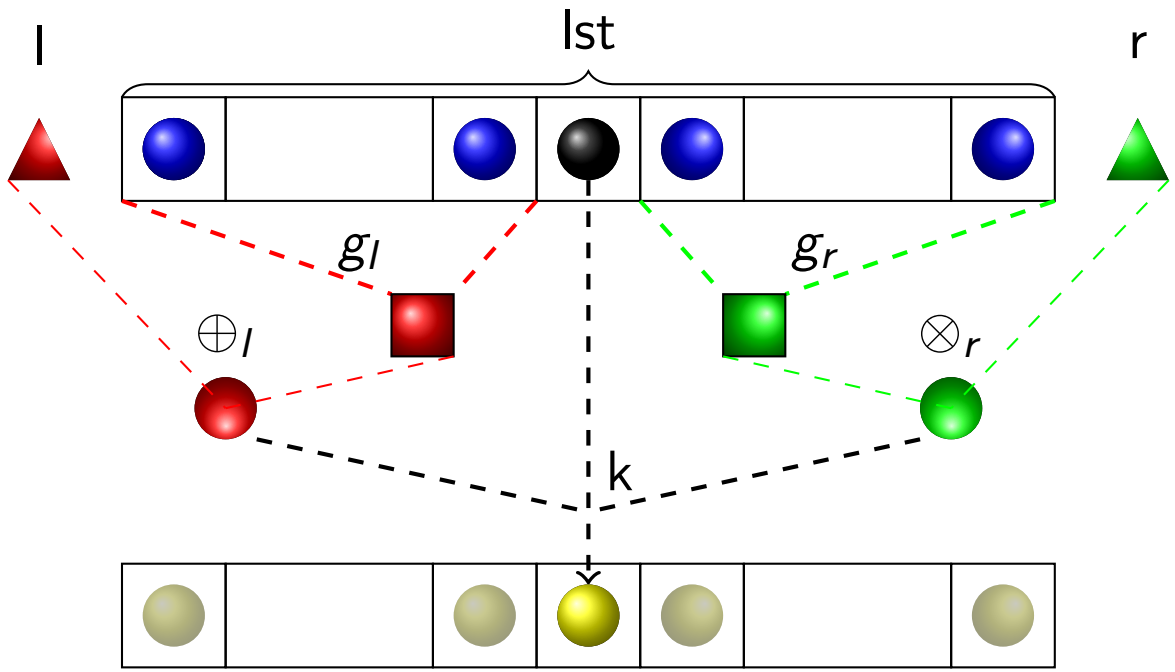


$$\forall\ \text{bh}\ l\ lst\ r\ i,\ nth\ i\ (\text{bh}\ l\ lst\ r) = k\ (l \oplus_l (gl\ (nth\ i\ (inits\ lst))))\ (nth\ i\ lst)\ (gr\ (nth\ i\ (tails\ lst))) \otimes_r r)$$

The BSP homomorphism - sequential computation

- *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*)

The BSP homomorphism - sequential computation

- *bh_comp*

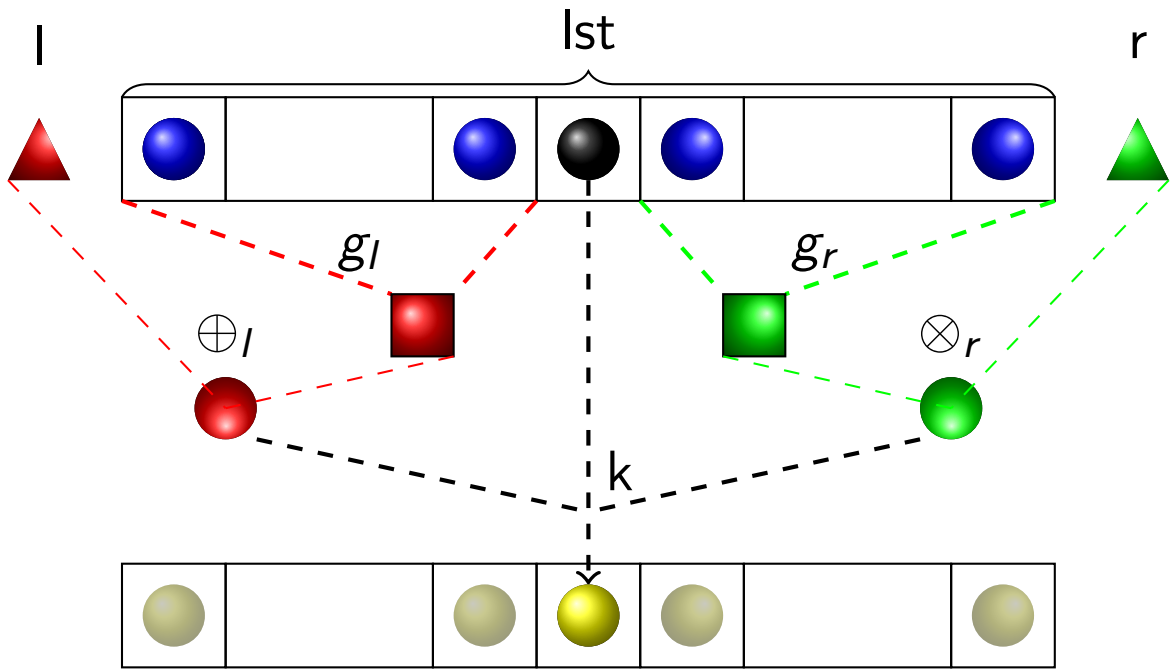
  ==

- *bh_seq*

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

- *bh_seq_opt*

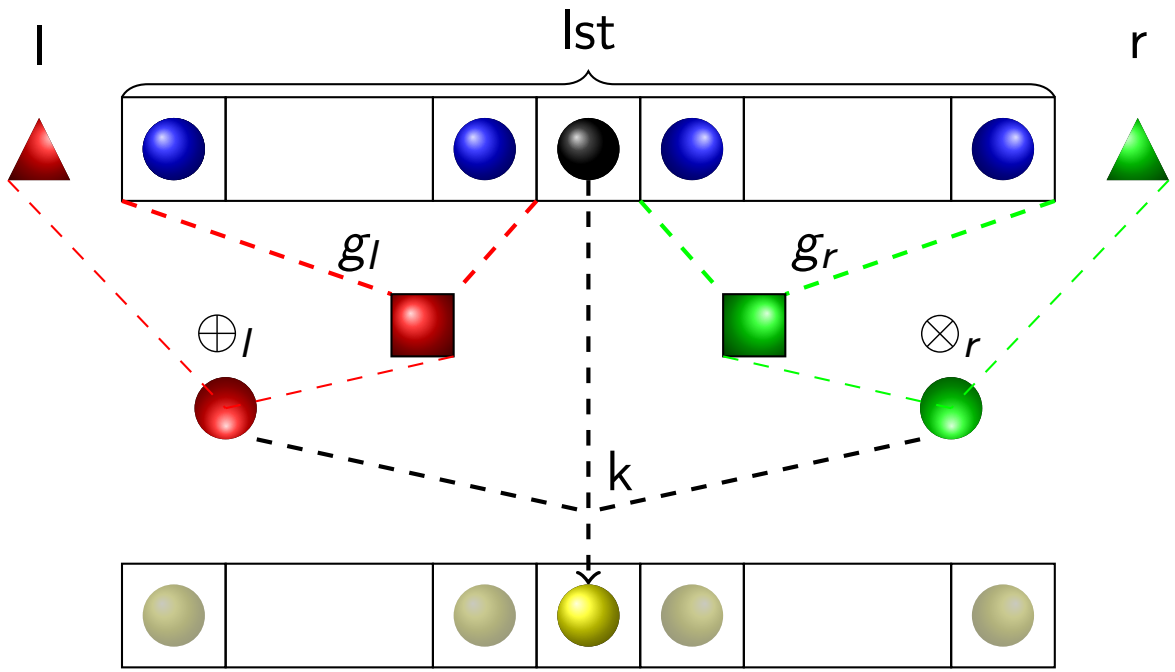
```
map3 k
  (fold_left ⊕_l l)
  l
  (fold_left ⊗_r l)
```



$\forall$ 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*)

The BSP homomorphism - sequential computation

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



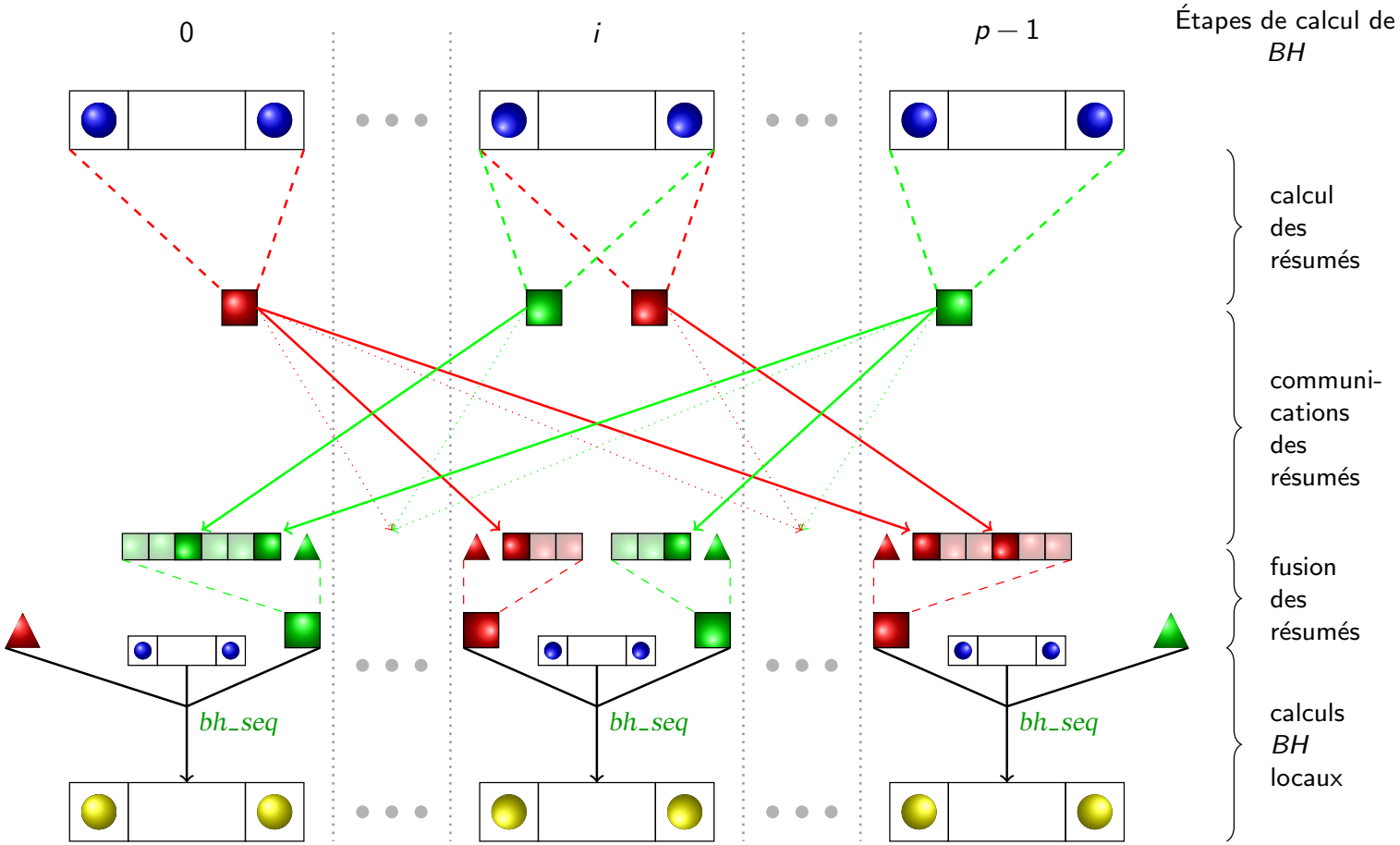
$$\forall\ \text{bh}\ l\ lst\ r\ i,\ nth\ i\ (\text{bh}\ l\ lst\ r) =$$
$$k$$
$$(l \oplus_l (gl\ (nth\ i\ (inits\ lst))))$$
$$(nth\ i\ lst)$$
$$(gr\ (nth\ i\ (tails\ lst))) \otimes_r r)$$

The BSP homomorphism - Model

```
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;
  }.
```

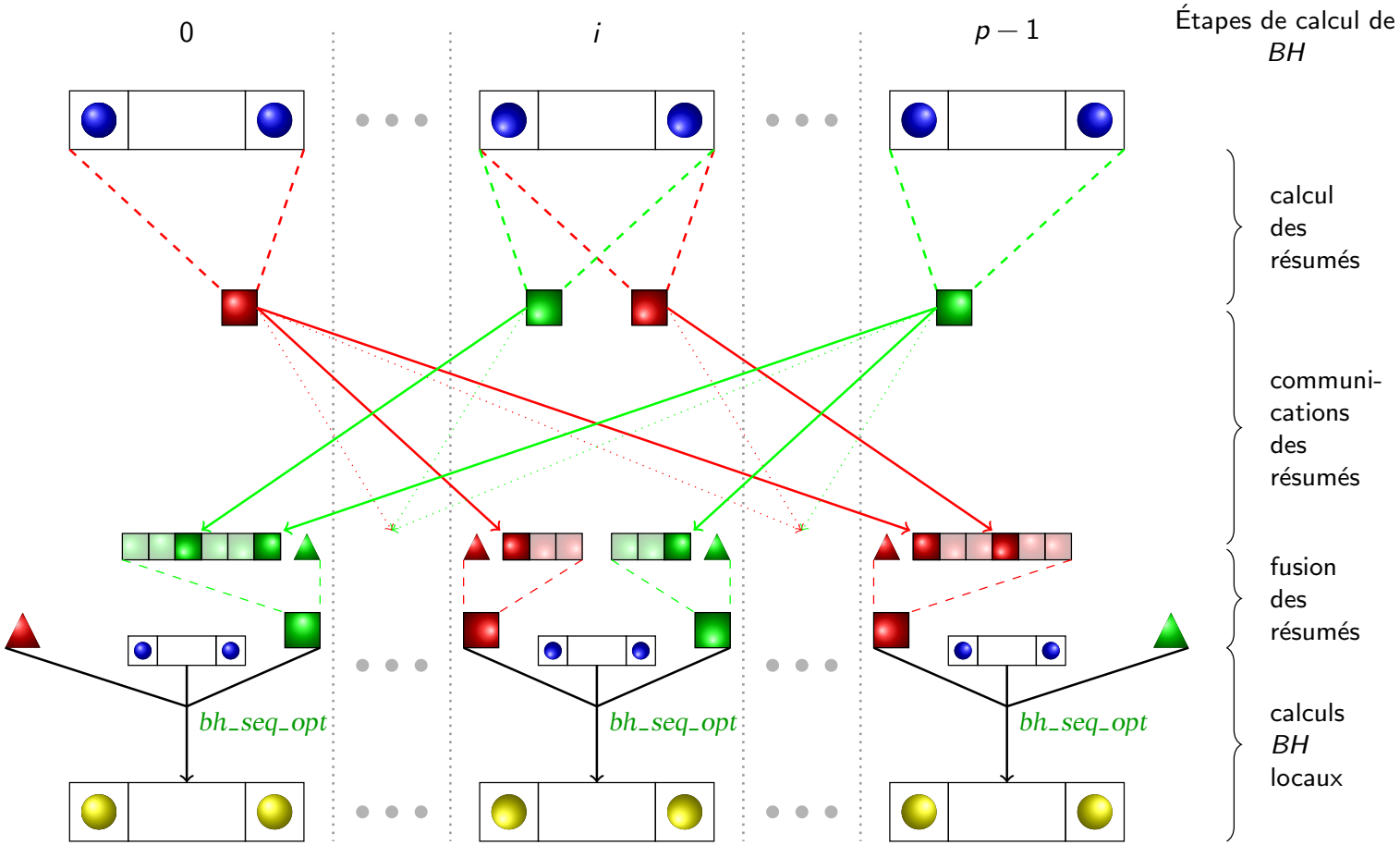
The BSP homomorphism - parallel computation

bh_bsml



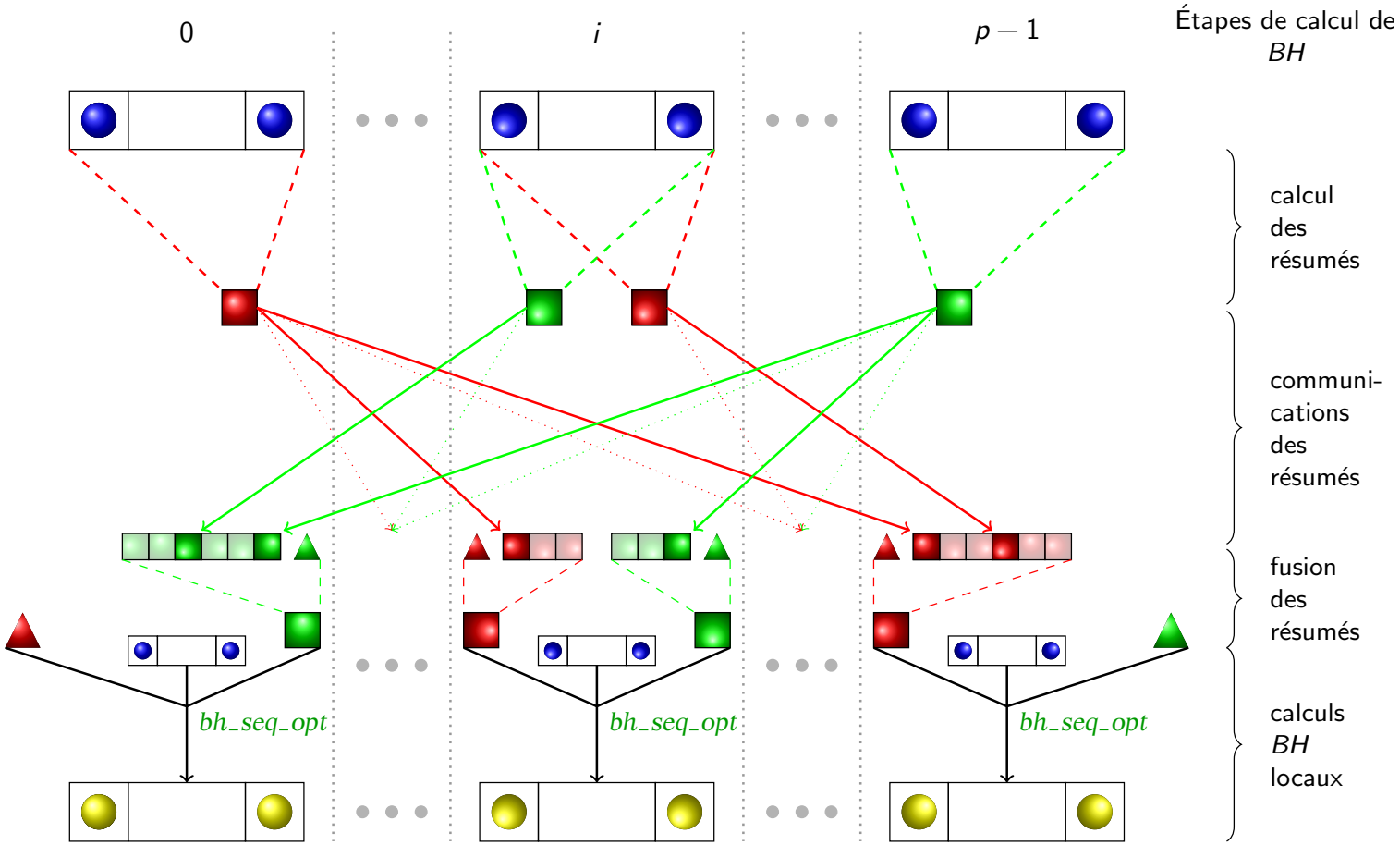
The BSP homomorphism - parallel computation

bh_bsml_opt



The BSP homomorphism - parallel computation

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



property of heat function

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

property of heat function

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

property of heat function

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

property of heat function

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

property of heat function

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

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

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

property of heat function

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

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

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

property of heat function

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

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

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

property of heat function

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

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

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

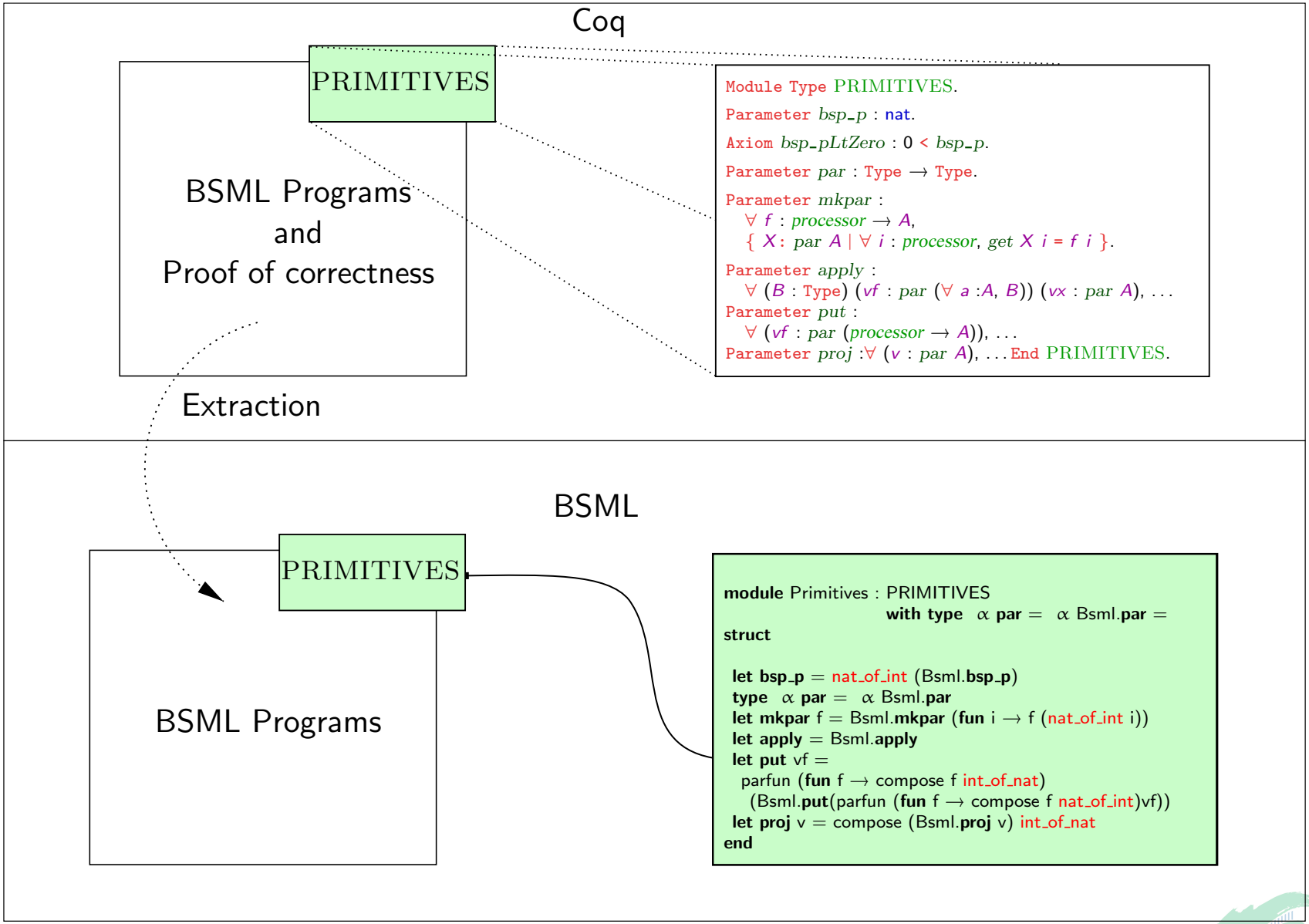
property of heat function

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

heatSeq *heatSeqBH* BH_PROP *BH_PROP_parallelizable*

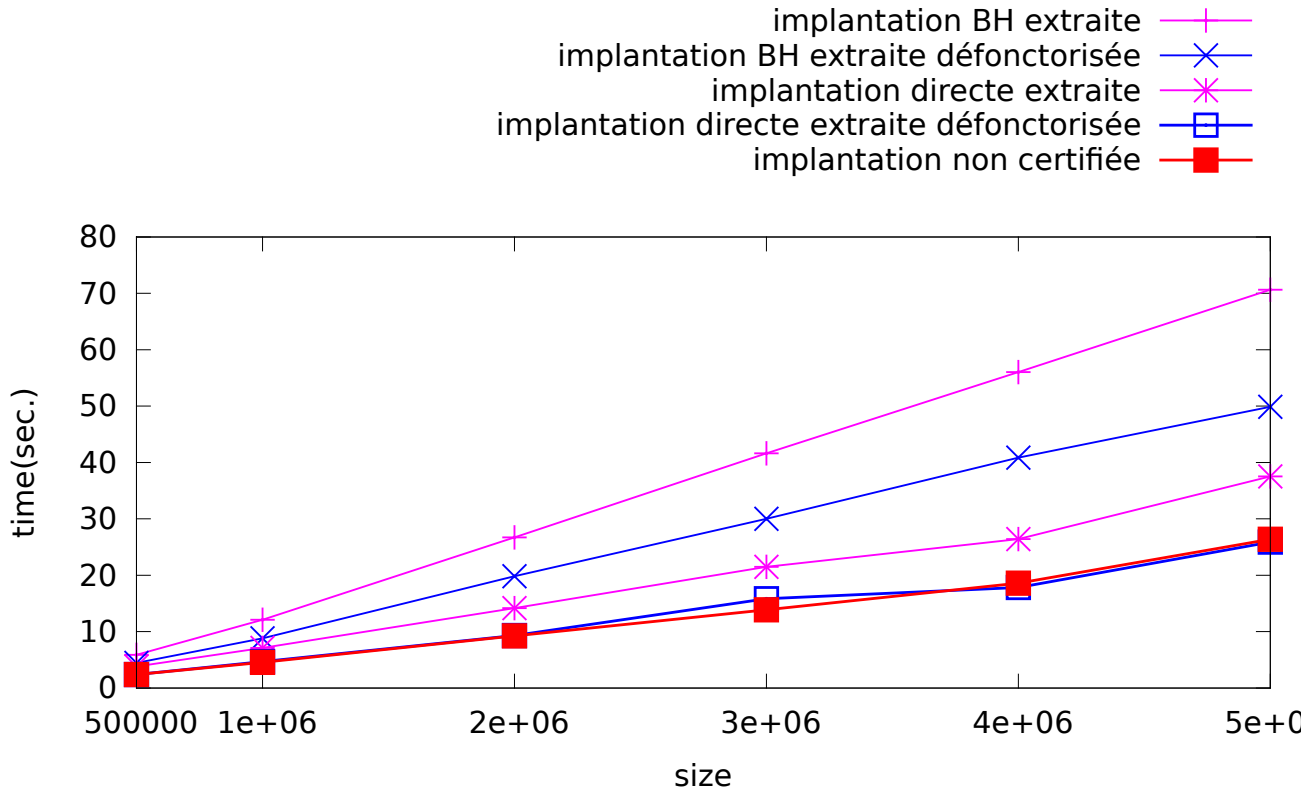
Parallel (*heatSeq*) (parallel (f := *heatSeqBH* ...))

Extraction


Coq
PRIMITIVES
BSML Programs
and
Proof of correctness
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
BSML Programs
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

Experimentation

- MIReV - PCs clusters
- 16 processors
- Un-functorisation to enable compiler optimisation



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

		Specifications	Proofs
Modelisation of BSML	Primitives	**30**	0
	Properties & stdlib	216	464
	Sequential implementation	60	35
Correct parallelisation		91	15
	list distribution	622	602
Skeletons	BH	456	884
	others (map,filter,last,...)	403	226
Applications	Heat equation	199	363
	Heat equation BH	186	**57**
	Counting	35	**0**
	tower building	105	59
	Maximum prefix sum	110	**0**
Sequential	LIFO's Coq library : lists, vector, algebra	1995	2827
Total		4508	5532

Conclusion

Development of BSML programs in Coq

- Programing style similar to usual BSML
- Extraction of parallel programs directly usable

Proof BSML programs in Coq

- Strongly specified programs
- correct and composable parallelisation

Algorithmic skeletons

- Eases the development of parallel programs (automates the parallelisation)
- Correctness proof done once and for all

Perspectives

BSML Primitives + **impérative style**

Reasonning on cost

Algorithmic Skeletons

New skeletons

Programs and correctness proofs

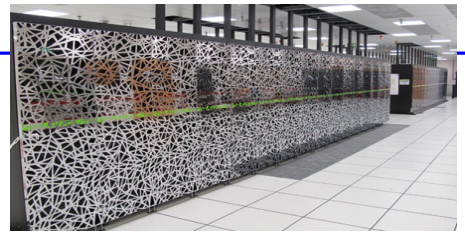
Extraction

Extracted BSML Programs

BSML primitives implementation

Un-functorisation and BSML compilation **verified**

Parallel Executions



Thank you for your attention
Questions ?

plan

1. Coq embedding of BSML
2. Correct parallelisation
3. Correct Programs Construction using Skeletons
4. Extraction and experimentation
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. ...
```

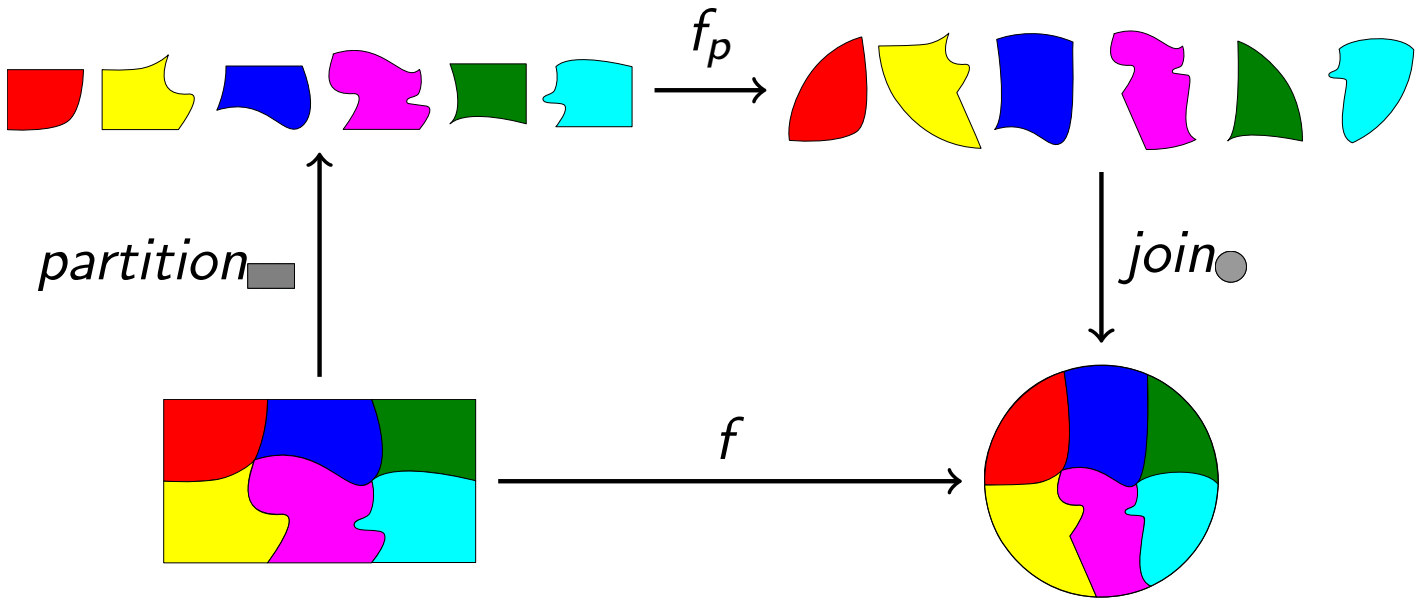
```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i =  get vr ( (i+dec) mod bsp_p)  } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p)  ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig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

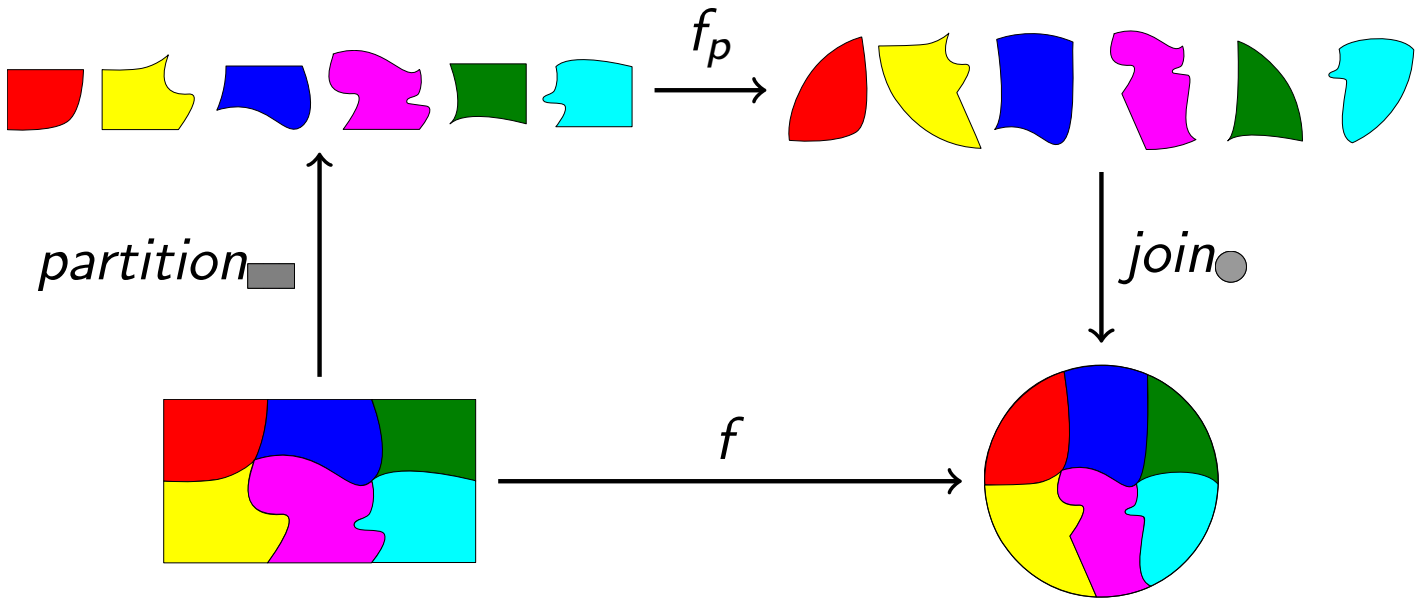
```
  Program Definition shift A dec (v : par A) :
    {vr:par A| ∀ i, get v i = get vr ( (i+dec) mod bsp_p) } :=
  let received := put
  (apply (mkpar (fun (i :processor) l (j :processor) ⇒
    if ( j==(( i + dec) mod bsp_p) ) then Some l else None
))
  v)
    in
  parfun (@noSome _)
    (parSig
  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

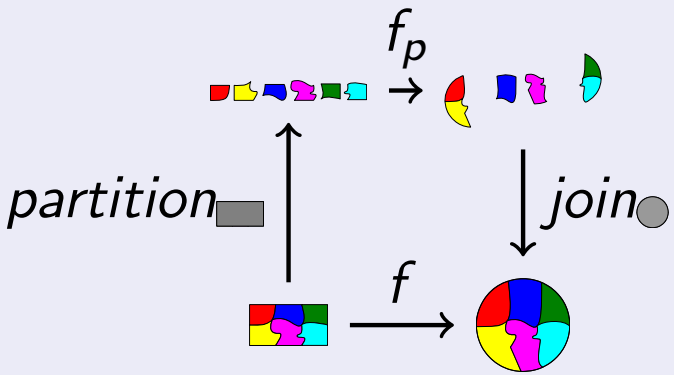
Parallélisation correcte

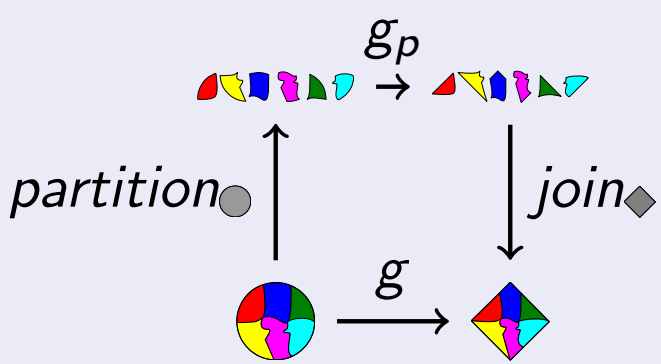


Parallélisation correcte

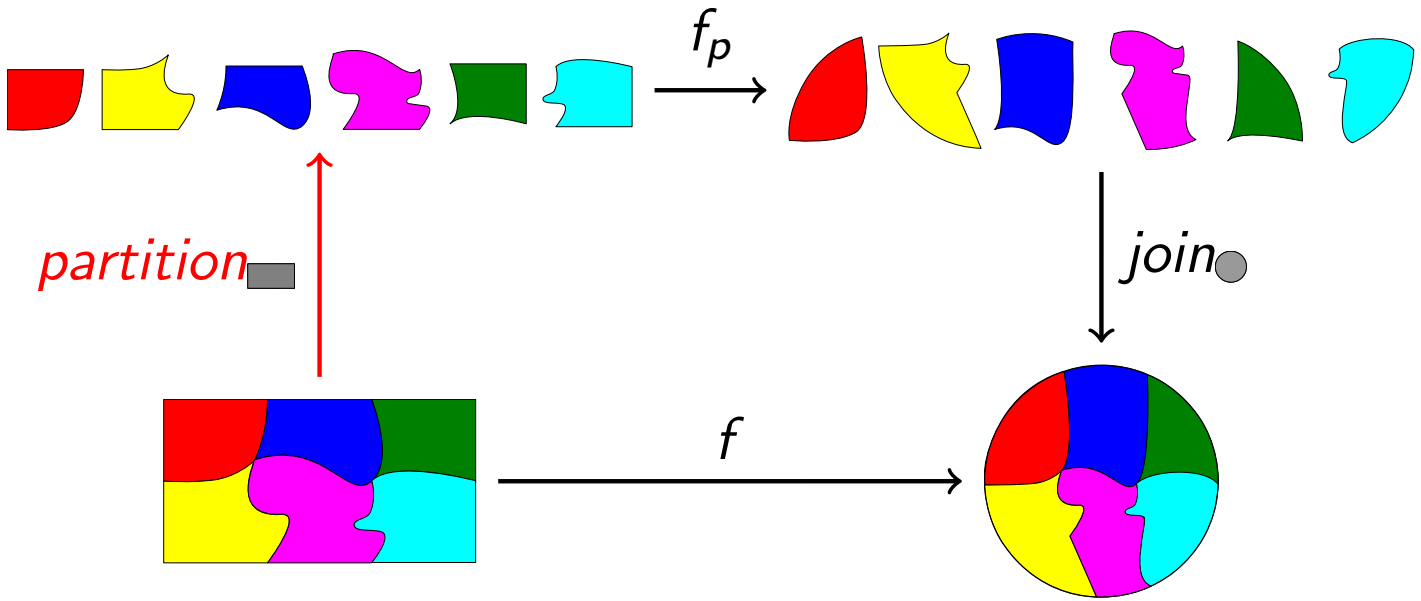


Composabilité : $g_p \circ f_p$?



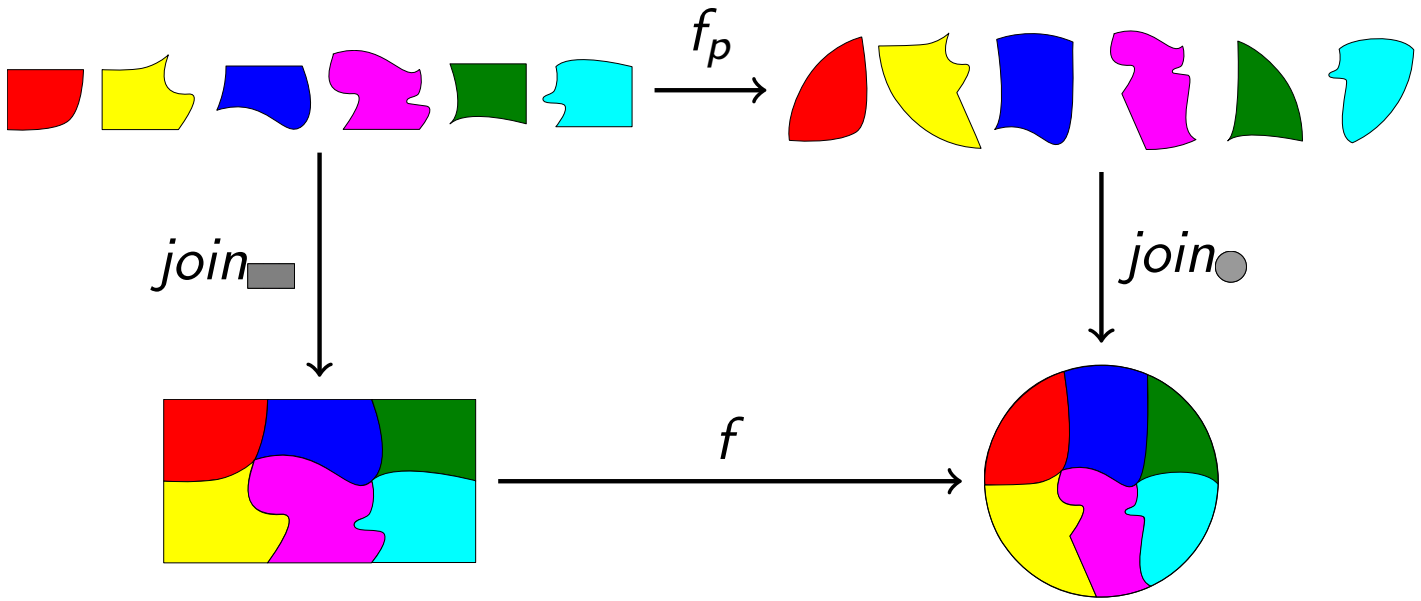


Parallélisation correcte

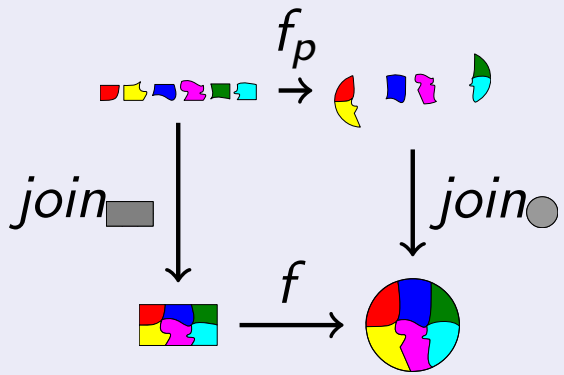


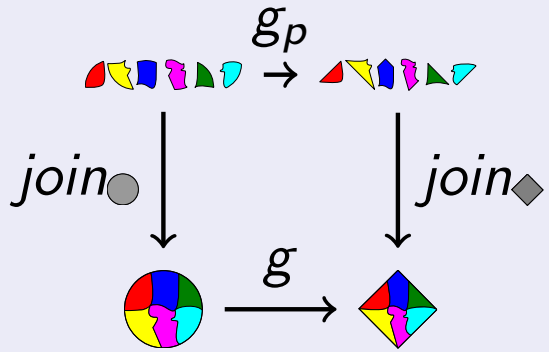
Composabilité : $g_p \circ f_p$?

Parallélisation correcte

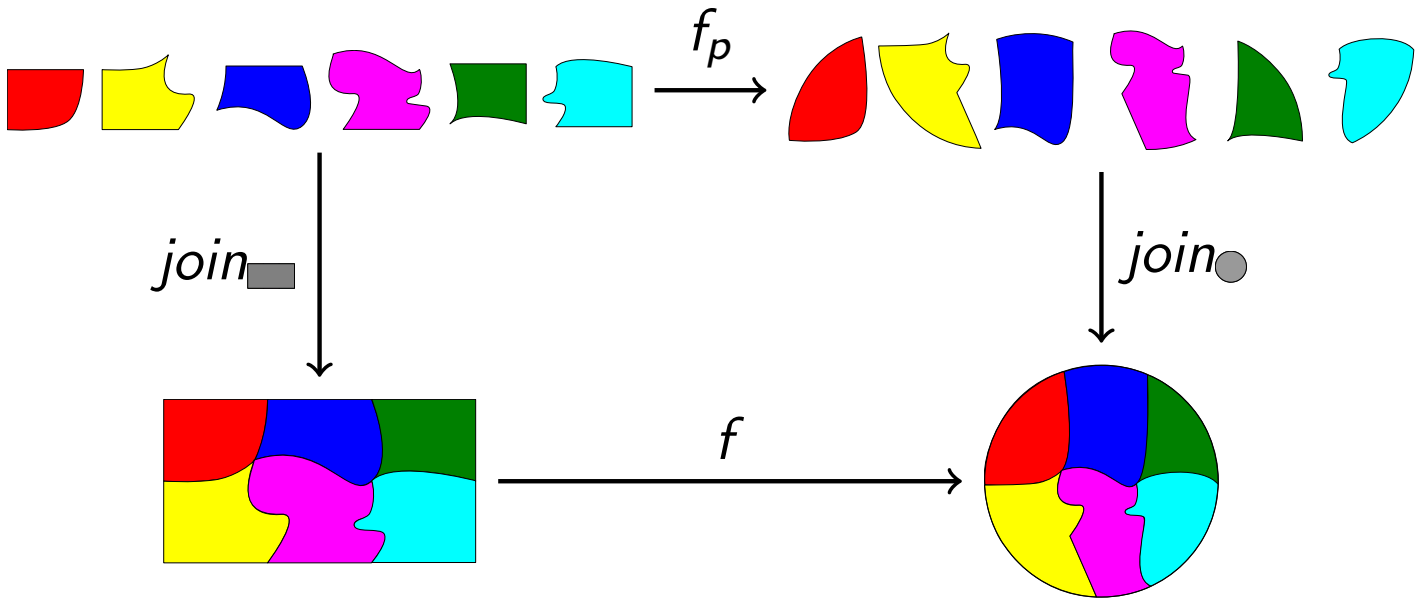


Composabilité : $g_p \circ f_p$?



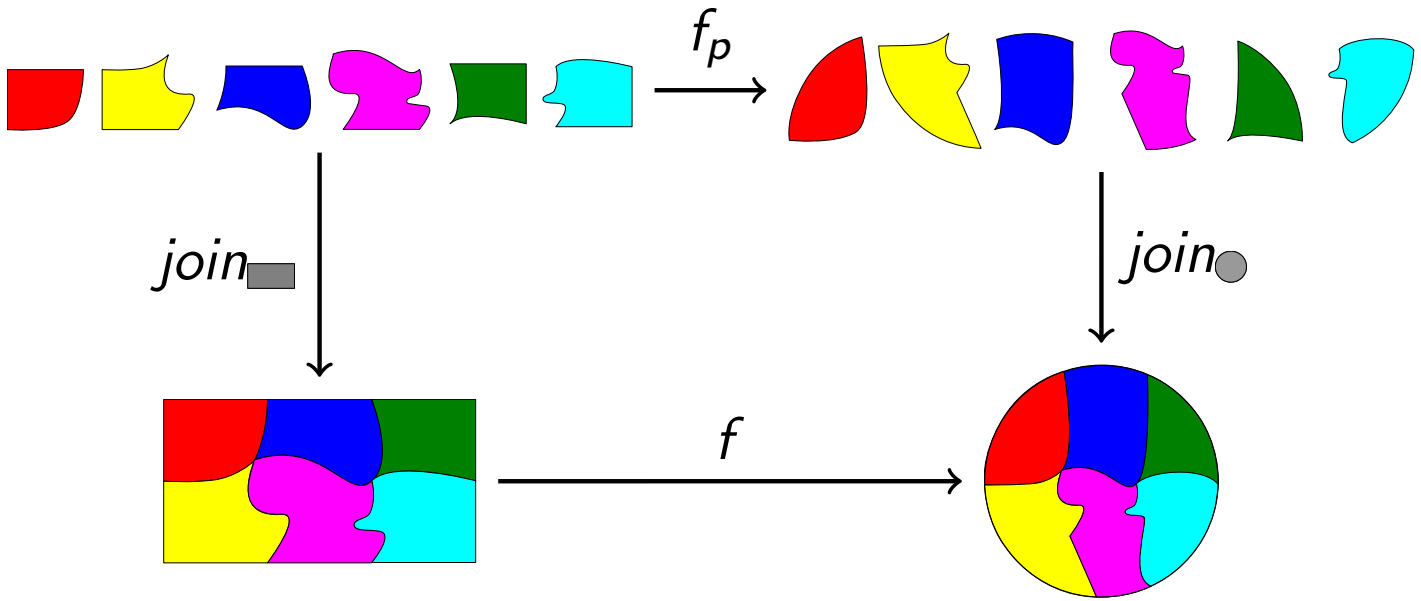


Parallélisation correcte

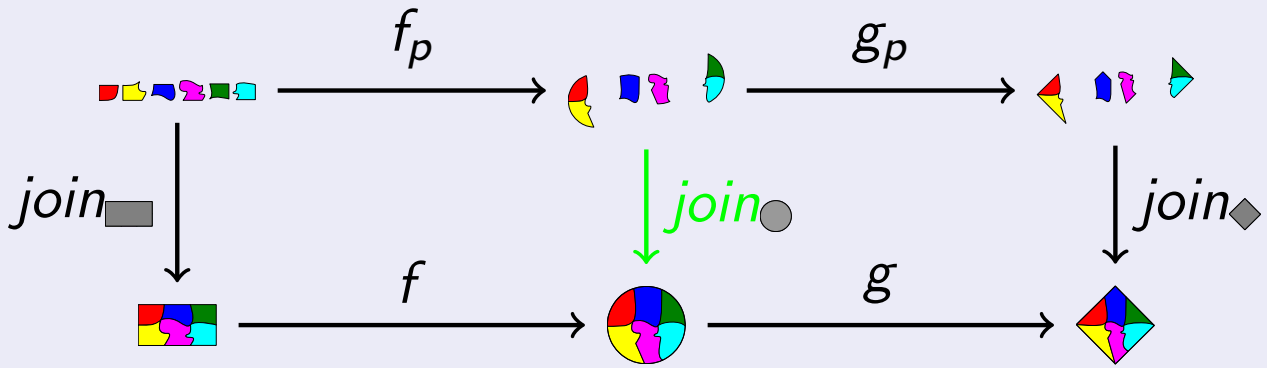


Composabilité : $g_p \circ f_p$?

Parallélisation correcte



Parallélisation correcte composable $g_p \circ f_p$



Fin