Formalisation d'une structure de donnée transiente et de ses itérateurs en logique de séparation

Séminaire Cambium

Alexandre Moine,
sous la direction d'Arthur Charguéraud et de François Pottier

30 août 2021

Sommaire

À propos de structures transientes

Une structure de donnée transiente est une structure qui offre :

- Une interface éphémère.
- Une interface persistante.
- Et des conversions efficaces entre les deux !

À propos de structures transientes

Une structure de donnée transiente est une structure qui offre :

- Une interface éphémère.
- Une interface persistante.
- Et des conversions efficaces entre les deux !

Par exemple Sek [1], une bibliothèque efficace pour des séquences transientes (push et pop des deux côtés, split, concat...).

```
opam install sek
```

1. Charguéraud and Pottier

À propos de structures transientes

Une structure de donnée transiente est une structure qui offre :

- Une interface éphémère.
- Une interface persistante.
- Et des conversions efficaces entre les deux !

Par exemple Sek [1], une bibliothèque efficace pour des séquences transientes (push et pop des deux côtés, split, concat...).

```
opam install sek
```

Ici, une version simplifiée de Sek : des piles transientes.

1. Charguéraud and Pottier

À propos de structures transientes

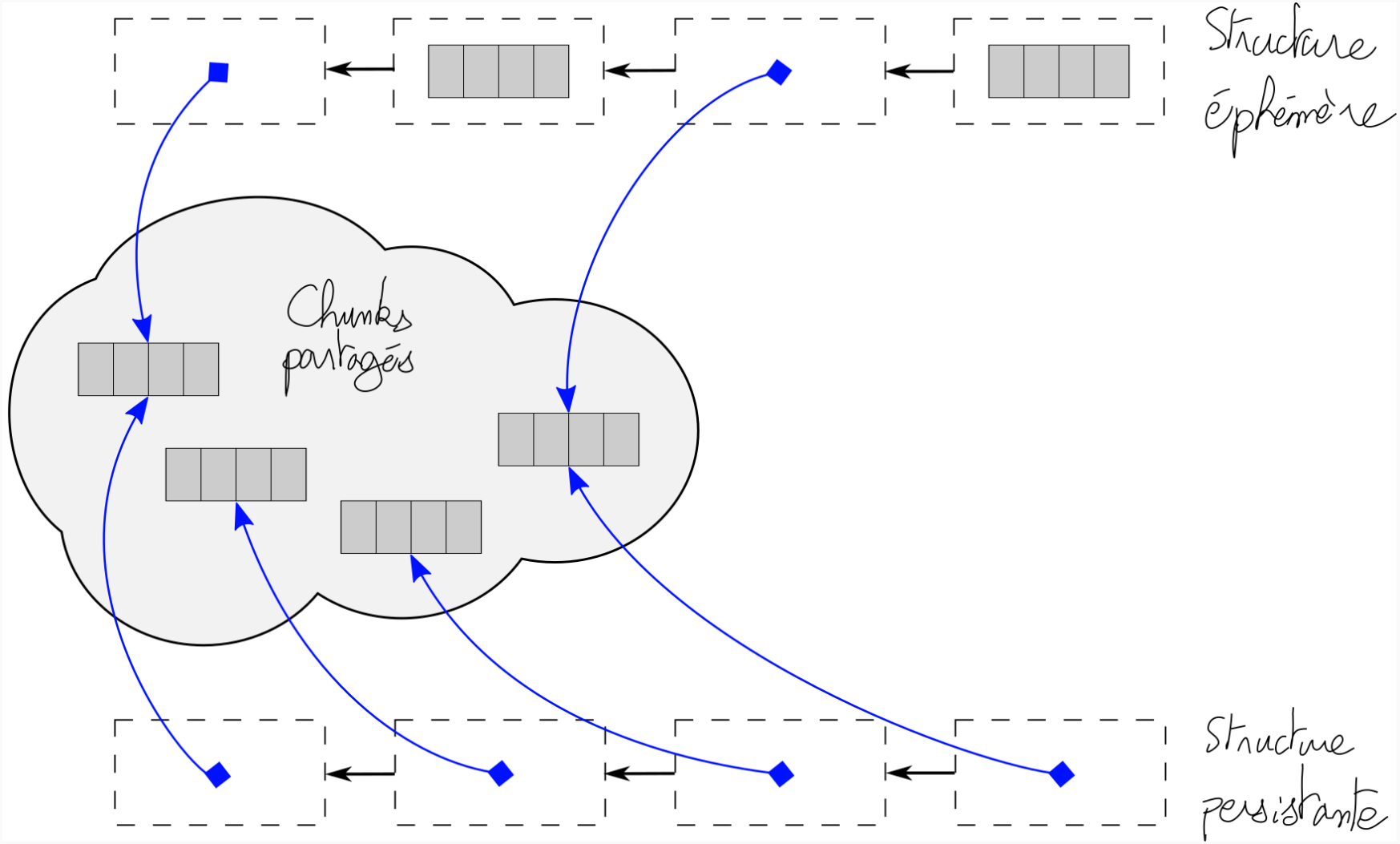
L'idée clé est d'utiliser des chunk sequences [1].

Un chunk est un tableau de capacité fixe.

- En les combinant astucieusement :
  $\rightarrow$ on obtient des structures de taille arbitraire.
- En les partageant :
  $\rightarrow$ on devient efficace !

1. Acar, Charguéraud, and Rainey (2014)

Un aperçu du chaudron magique



Cet exposé

On voudrait :

1. Des preuves de correction fonctionnelle.
2. Des preuves de complexité en temps.

Cet exposé

On voudrait :

1. Des preuves de correction fonctionnelle.
2. Des preuves de complexité en temps.

Pour ça, nous allons utiliser :

1. La logique de séparation, avec CFML [1].
2. Les crédits temps [2].

1. Charguéraud (2011)
2. Charguéraud and Pottier (2019)

Sommaire

Les chunks éphémères

Notre objet de base est donc un chunk, dit éphémère :

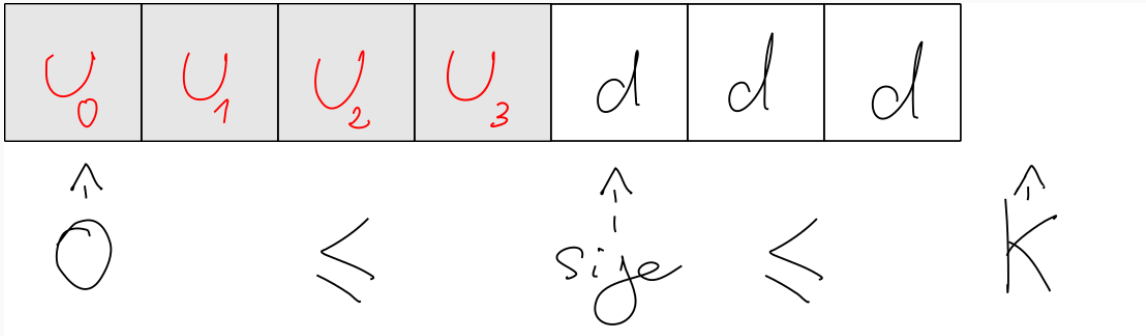
```
type 'a echunk = {
  data : 'a array; (* Un tableau de taille K *)
  mutable size : int;
  default : 'a (* Pour les cases vides *) }
```

Les chunks éphémères

Notre objet de base est donc un chunk, dit éphémère :

```
type 'a echunk = {
  data : 'a array; (* Un tableau de taille K *)
  mutable size : int;
  default : 'a (* Pour les cases vides *) }
```

Par exemple, une pile $u_3 :: u_2 :: u_1 :: u_0 :: nil$

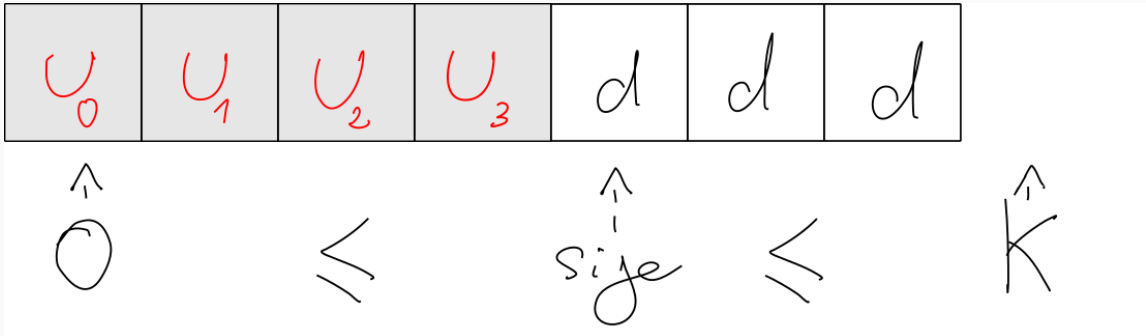


Les chunks éphémères

Notre objet de base est donc un chunk, dit éphémère :

```
type 'a echunk = {
  data : 'a array; (* Un tableau de taille K *)
  mutable size : int;
  default : 'a (* Pour les cases vides *) }
```

Par exemple, une pile $u_3 :: u_2 :: u_1 :: u_0 :: nil$



Les chunks éphémères offrent d'empiler et de dépiler en $O(1)$.

Les chunks partageables

Approche naïve pour la persistance : tout copier tout le temps.

Les chunks partageables

Approche naïve pour la persistance : tout copier tout le temps.

On peut faire mieux

```
type 'a schunk = {
  support : 'a echunk;
```

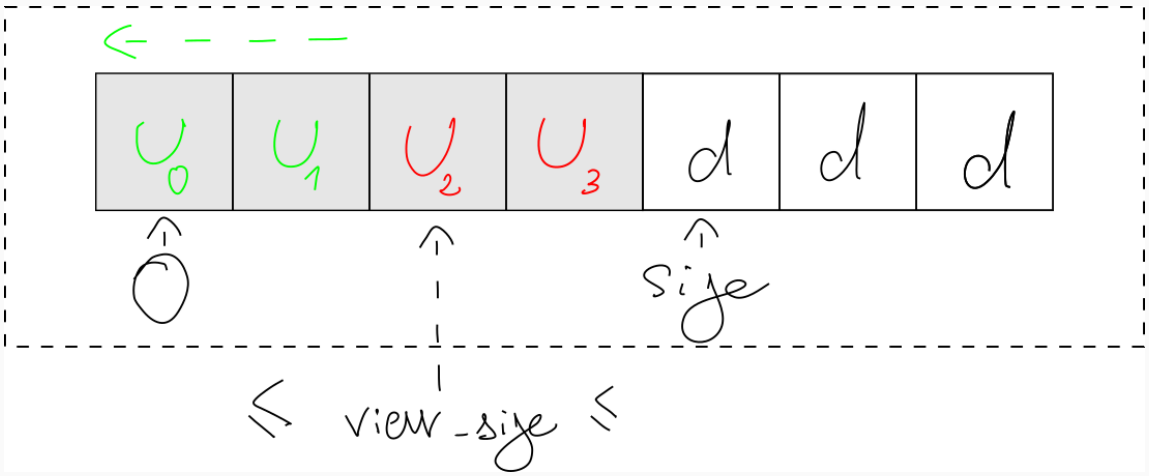
Les chunks partageables

Approche naïve pour la persistance : tout copier tout le temps.

On peut faire mieux avec des vues...

```
type 'a schunk = {
  support : 'a echunk;
  view_size : int;
```

Par exemple, une pile $u_1 :: u_0 :: nil$



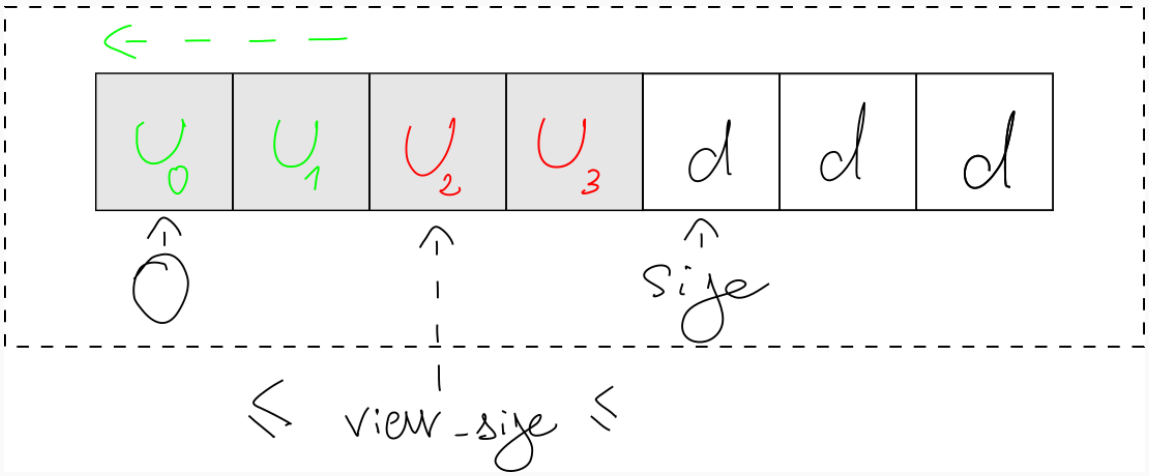
Les chunks partageables

Approche naïve pour la persistance : tout copier tout le temps.

On peut faire mieux avec des vues... et de la possession !

```
type 'a schunk = {
  support : 'a echunk;
  view_size : int;
  owner : Id.t }
```

Par exemple, une pile $u_1 :: u_0 :: nil$



À propos de possession

- Chaque chunk partageable a un identifiant.
- Cet identifiant est celui de son propriétaire.
- Identifiants égaux $\iff$ possession unique.
- Possession unique $\implies$ écriture sans effet de bord.
- Partage en $O(1)$: génération d'un nouvel identifiant.

Les chunks partageables, et leur interface

Les chunks partageables permettent :

- D'empiler :
  - Si possession unique[1], on écrit dans le support en $O(1)$.
  - Sinon copy-on-write en $O(K)$.
- De dépiler en $O(1)$.

1. ou situation favorable

Les piles éphémères

Une chunk sequence pour représenter une pile éphémère.

```
type 'a estack = {
  mutable front : 'a echunk;
  mutable tail  : 'a schunk list;
  mutable id    : Id.t }
```

- Tous les champs sont mutables.

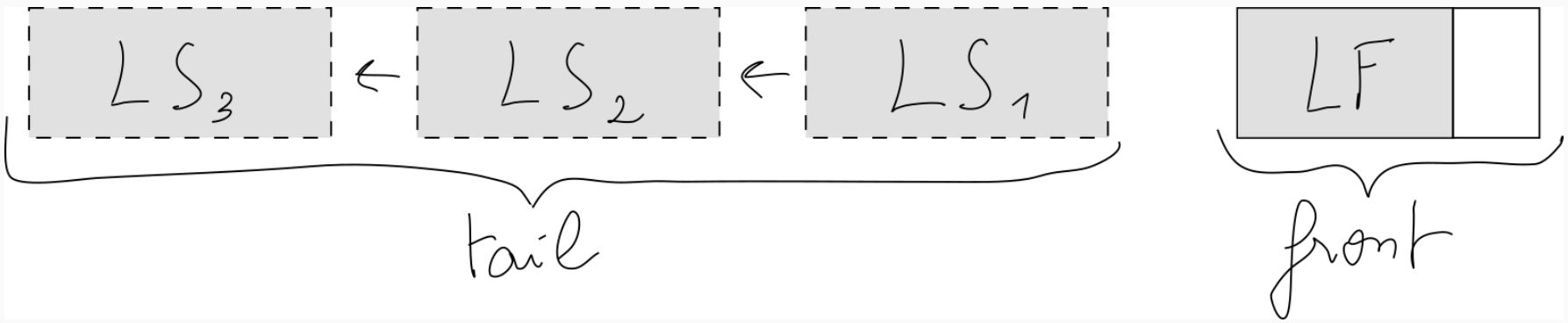
Les piles éphémères

Une chunk sequence pour représenter une pile éphémère.

```
type 'a estack = {
  mutable front : 'a echunk;
  mutable tail  : 'a schunk list;
  mutable id    : Id.t }
```

- Tous les champs sont mutables.

Pour représenter une pile `LF` ++ `concat` `LS`



Les piles éphémères, et leur interface

Les piles éphémères offrent :

1. D'empiler en temps constant amorti.

Les piles éphémères, et leur interface

Les piles éphémères offrent :

1. D'empiler en temps constant amorti.
   $\rightarrow$ allocation de chunk vide, de temps en temps [1].

1. En fait, on fait attention à ne pas désallouer trop vite les chunks vides.

Les piles éphémères, et leur interface

Les piles éphémères offrent :

1. D'empiler en temps constant amorti.
   $\rightarrow$ allocation de chunk vide, de temps en temps [1].
2. De dépiler en temps constant amorti.

1. En fait, on fait attention à ne pas désallouer trop vite les chunks vides.

Les piles éphémères, et leur interface

Les piles éphémères offrent :

1. D'empiler en temps constant amorti.
   $\rightarrow$ allocation de chunk vide, de temps en temps [1].
2. De dépiler en temps constant amorti.
   $\rightarrow$ copie d'un schunk non possédé, de temps en temps.

1. En fait, on fait attention à ne pas désallouer trop vite les chunks vides.

Les piles persistantes

```
type 'a pstack = {
  pfront : 'a schunk;
  ptail  : 'a schunk list }
```

- Le type est très similaire à celui des piles éphémères.
- C'est une structure immutable !
- Tous les chunks sont partagés.

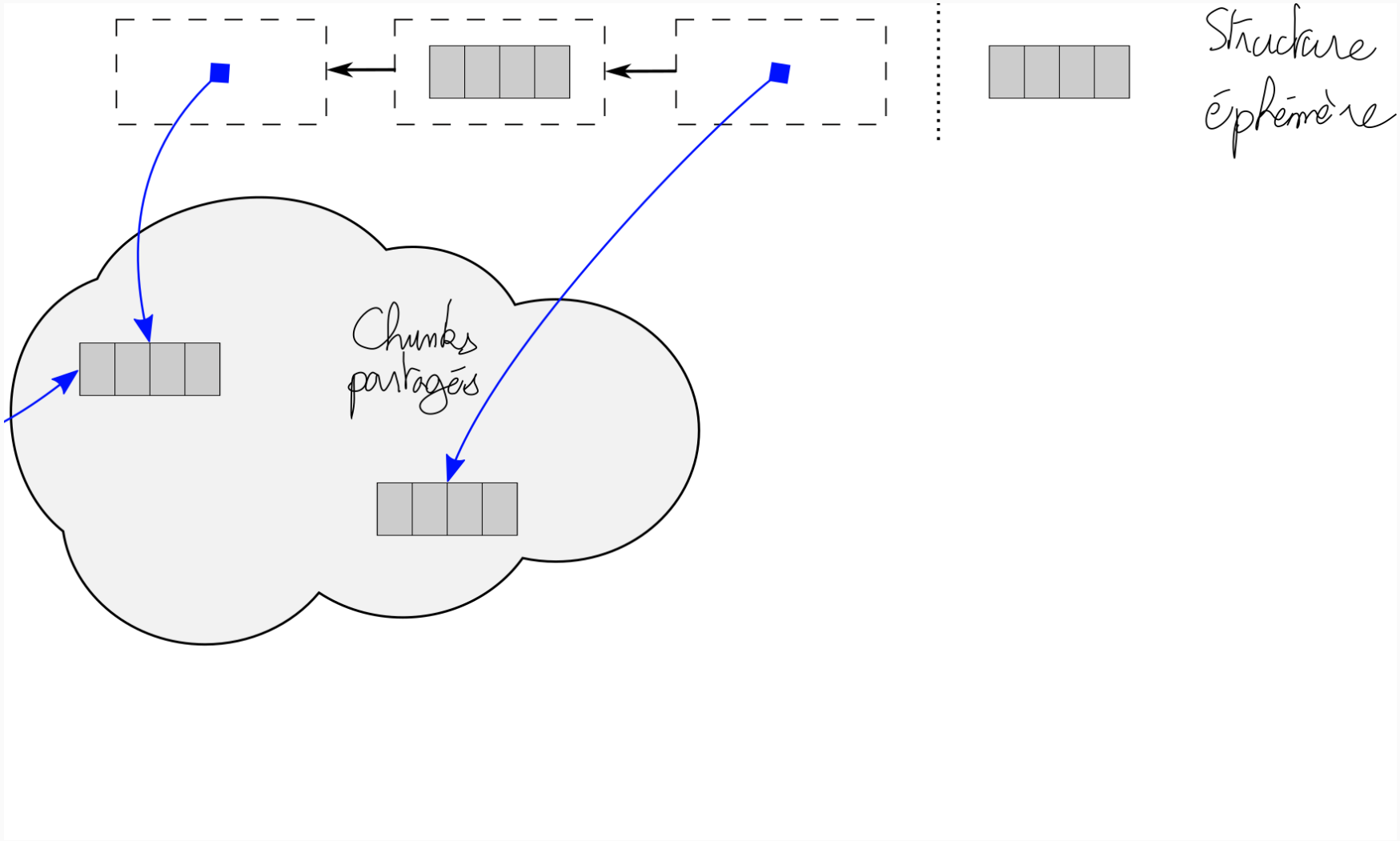
Les piles persistantes, et leur interface

Les piles persistantes offrent :

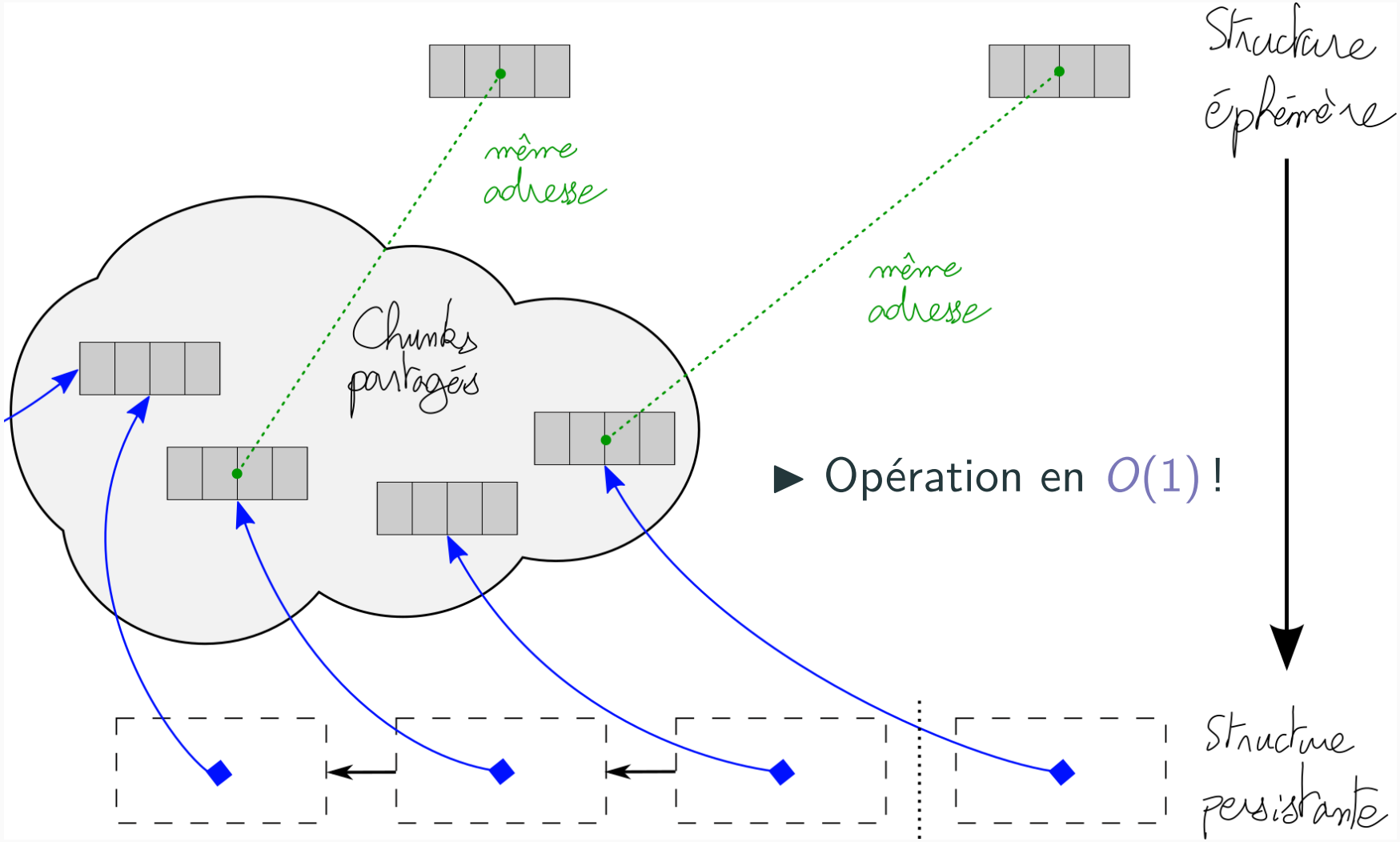
- D'empiler :
  - Si cas favorable[1], on écrit dans le support en $O(1)$.
  - Sinon copy-on-write en $O(K)$.
- De dépiler en $O(1)$.

1. Attention, pas d'argument de possession possible ici.

Conversion : prendre un instantané

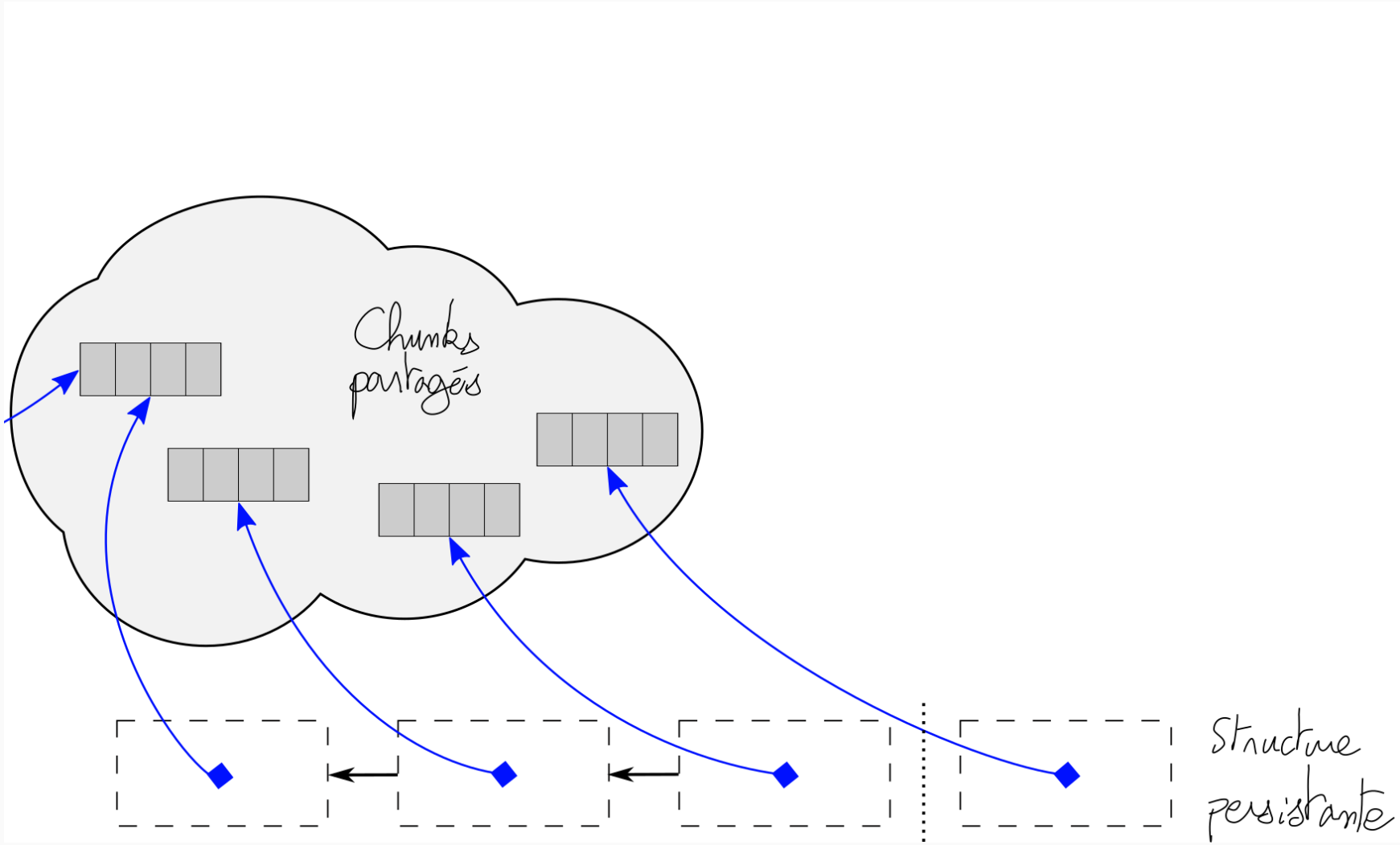


Conversion : prendre un instantané

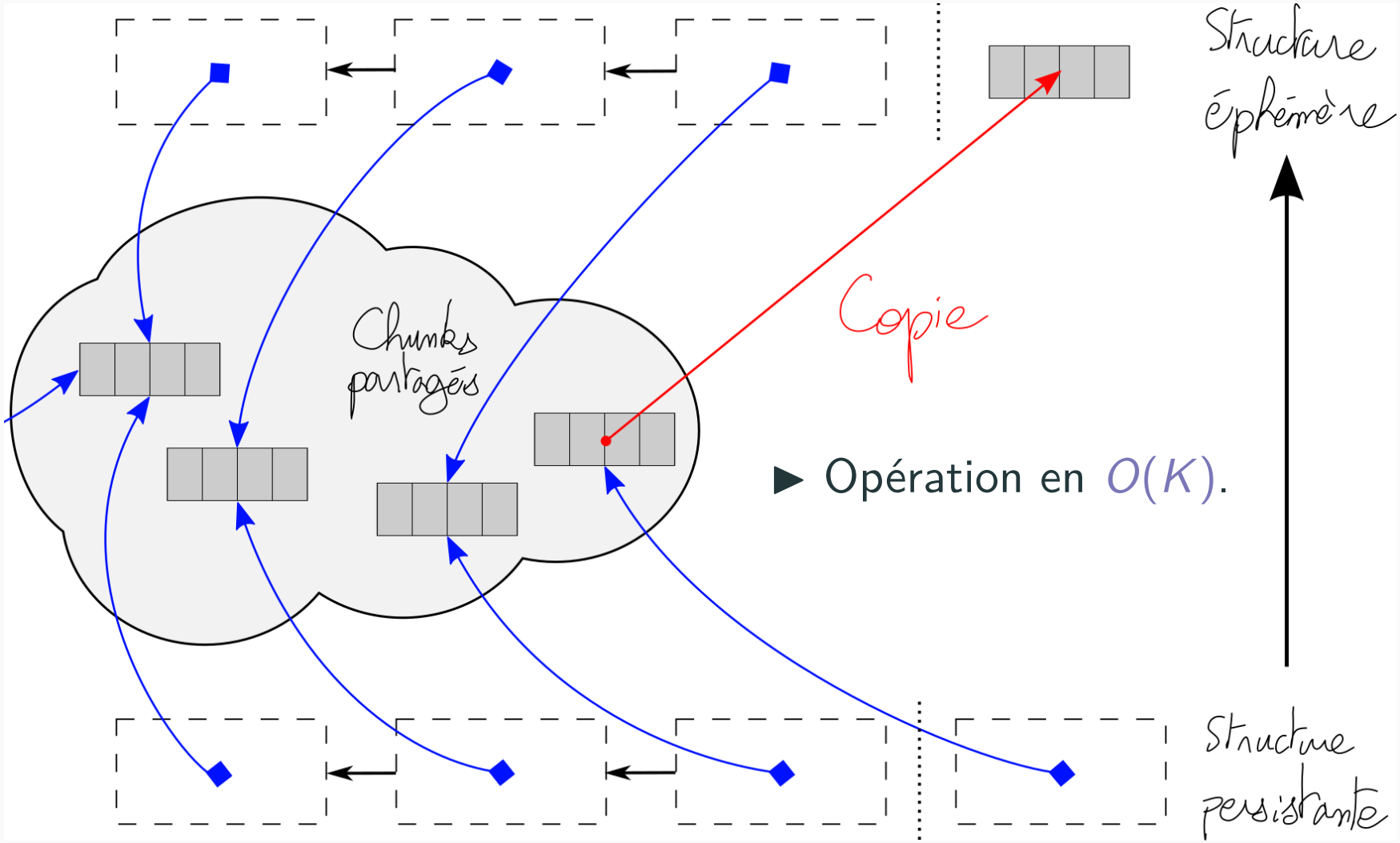


- Opération en $O(1)$!

Conversion : révéler la mutabilité



Conversion : révéler la mutabilité



- Opération en $O(K)$.

Des itérateurs

```
type 'a iterator

val iter_on_estack : 'a estack -> 'a iterator
val iter_on_pstack : 'a pstack -> 'a iterator
```

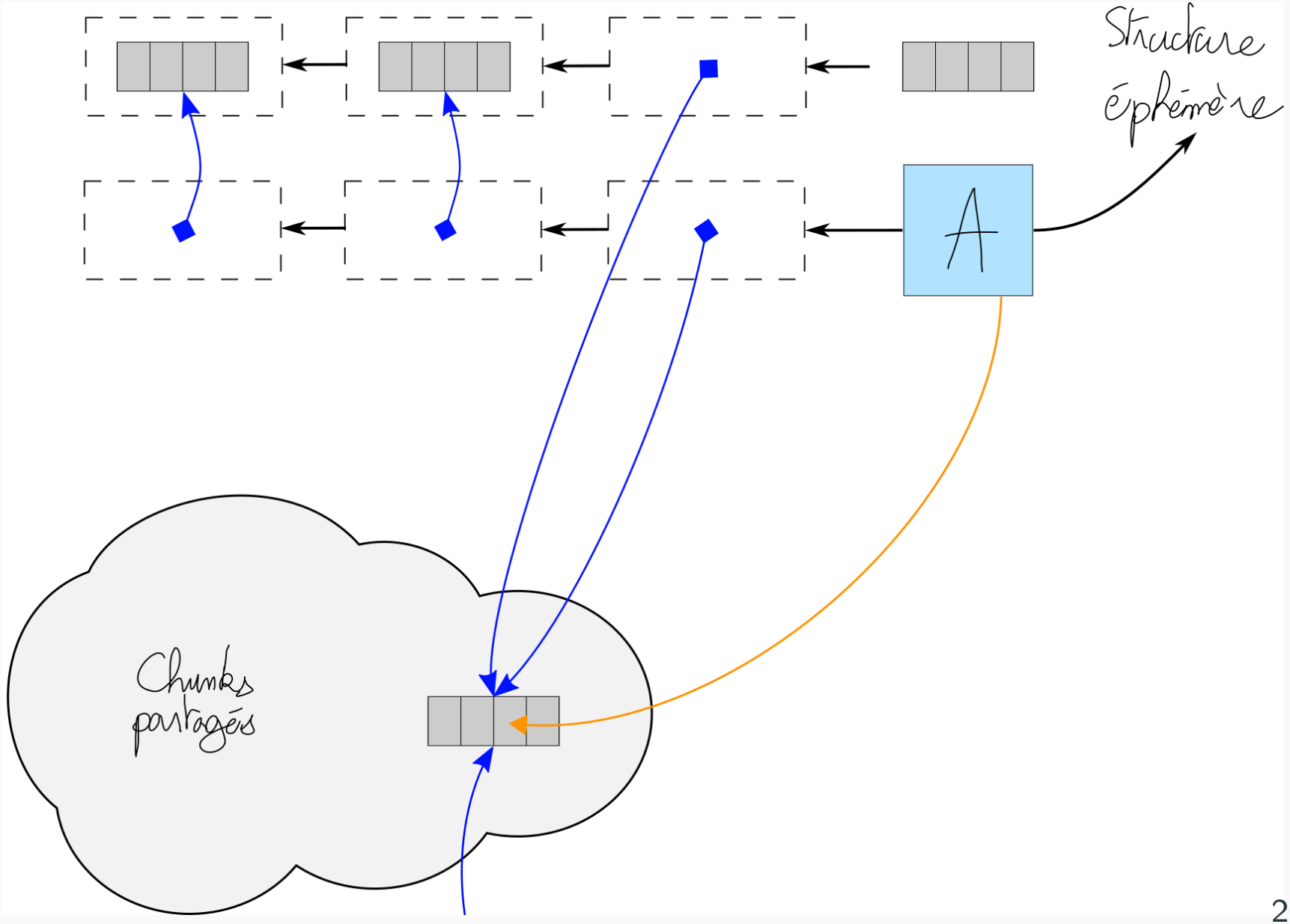
Des itérateurs

```
type 'a iterator

val iter_on_estack : 'a estack -> 'a iterator
val iter_on_pstack : 'a pstack -> 'a iterator

val get      : 'a iterator -> 'a
val move     : 'a iterator -> unit
val finished : 'a iterator -> bool
```

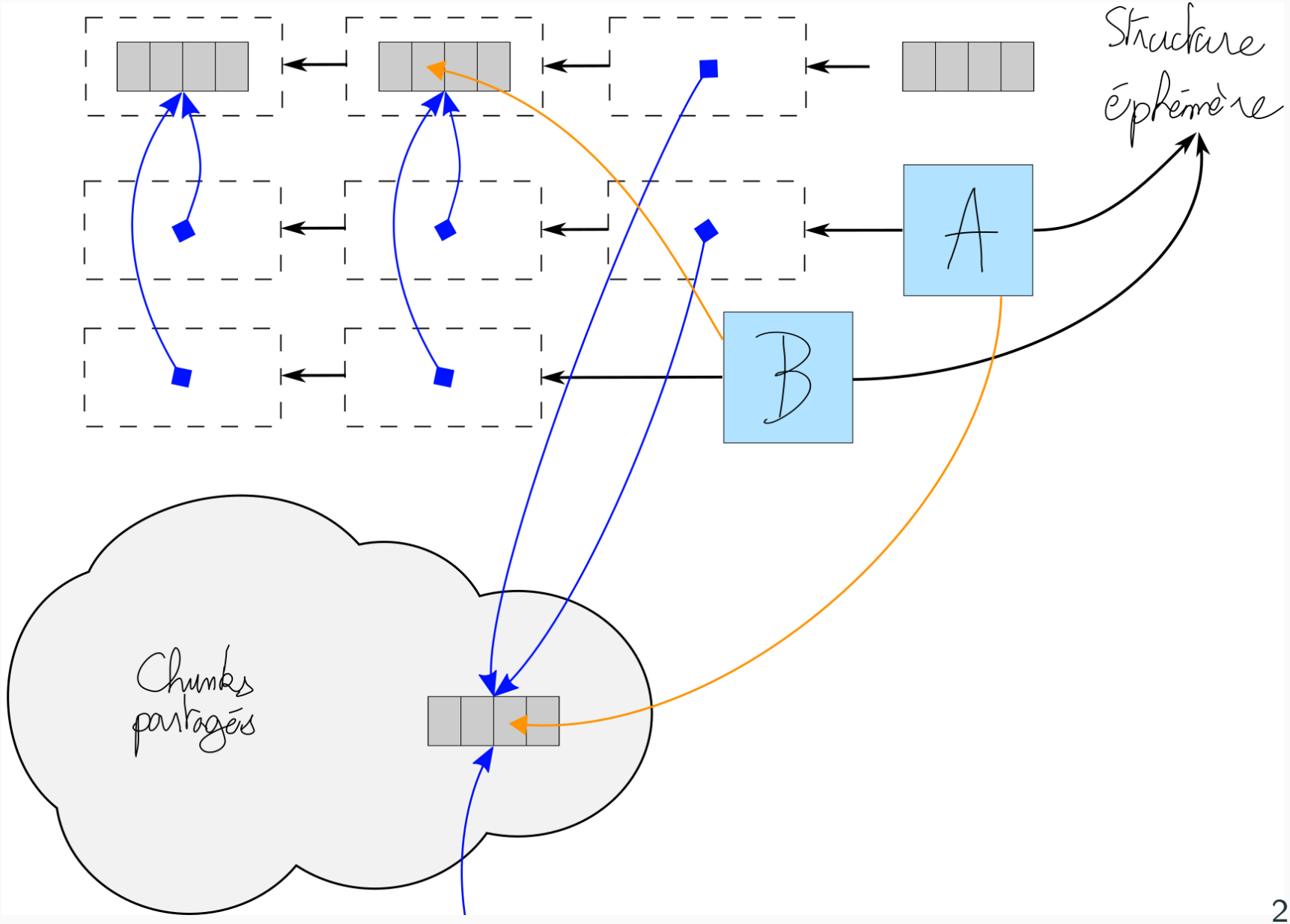
Des itérateurs

```
type 'a iterator

val iter_on_estack : 'a estack -> 'a iterator
val iter_on_pstack : 'a pstack -> 'a iterator

val get      : 'a iterator -> 'a
val move     : 'a iterator -> unit
val finished : 'a iterator -> bool

(* Valid only if the underlying stack is ephemeral *)
val set      : 'a iterator -> 'a -> unit
```

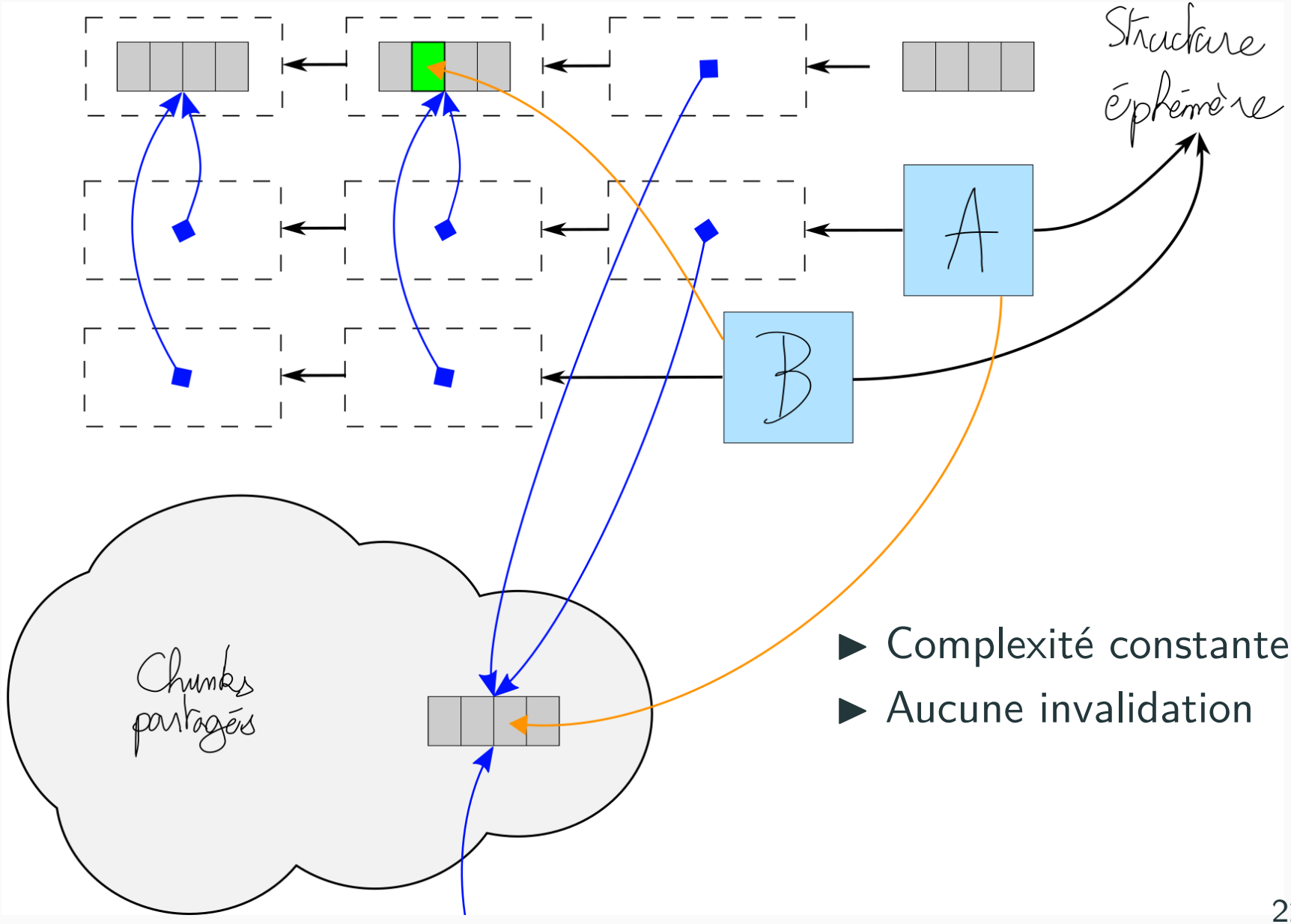
Aperçu d'un itérateur



`set` : le cas des chunks possédés uniquement

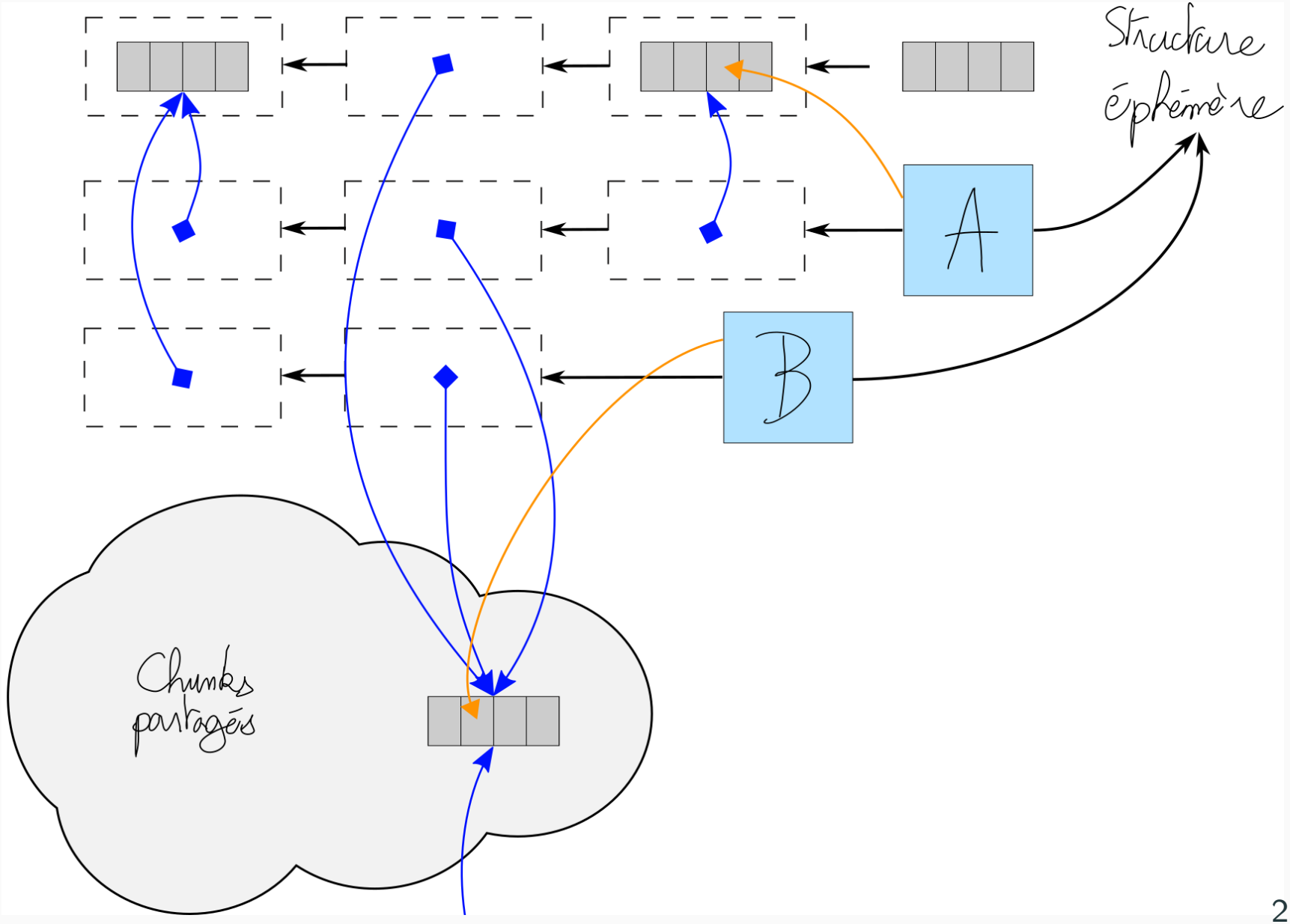


`set` : le cas des chunks possédés uniquement

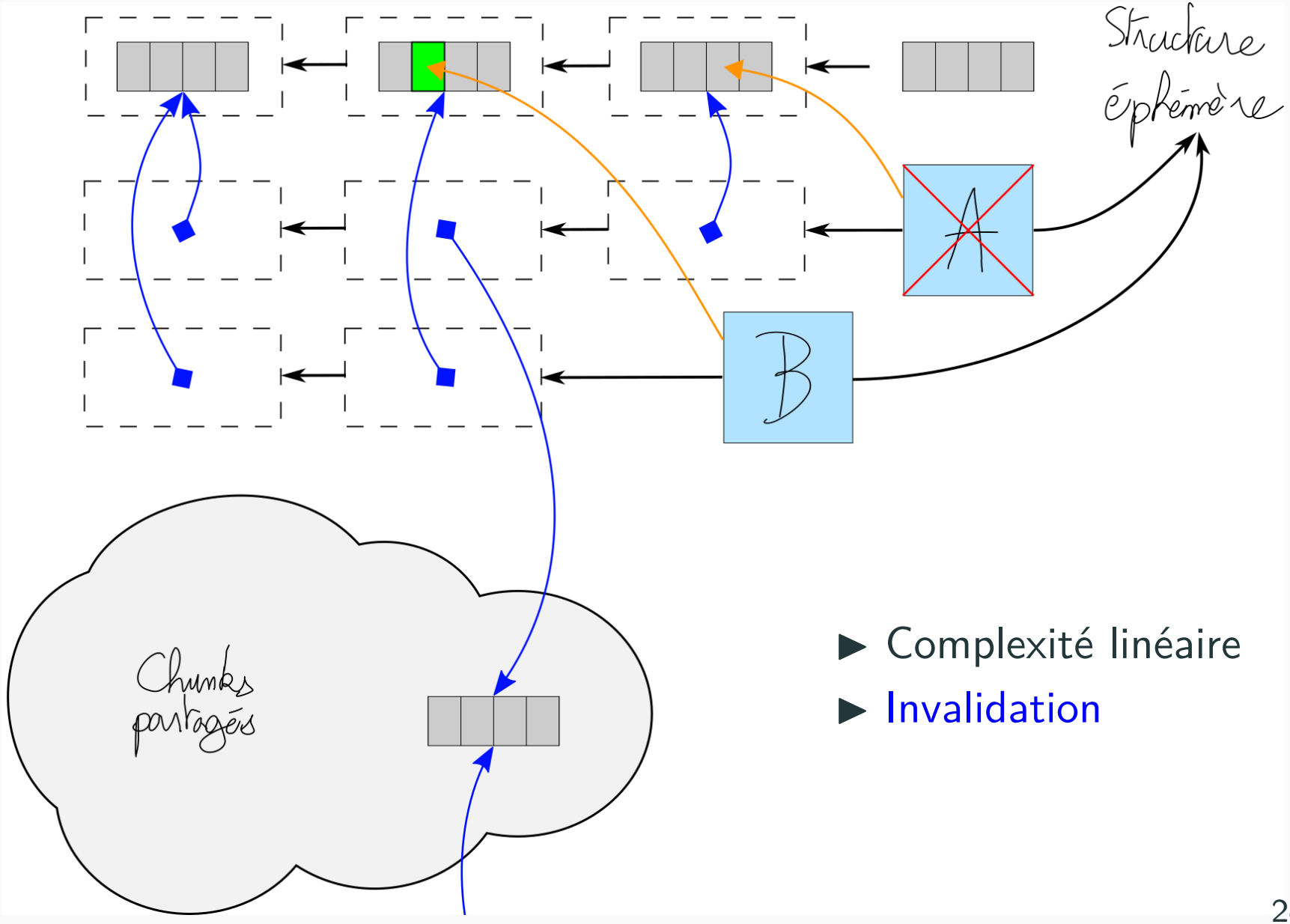


- Complexité constante
- Aucune invalidation

`set` : le cas des chunks partagés



`set` : le cas des chunks partagés



- Complexité linéaire
- Invalidation

Bilan d'étape

Plusieurs structures s'imbriquent :

- EChunk (chunk éphémère)
- SChunk (chunk partageable)
- Itérateur
- EStack (pile éphémère)
- PStack (pile persistante)

Bilan d'étape

Plusieurs structures s'imbriquent :

- EChunk (chunk éphémère)
- SChunk (chunk partageable)
- Itérateur
- EStack (pile éphémère)
- PStack (pile persistante)

Ainsi que plusieurs notions (au cœur de Sek) :

- Possession
- Complexité amortie

Bilan d'étape

Plusieurs structures s'imbriquent :

- EChunk (chunk éphémère)
- SChunk (chunk partageable)
- Itérateur
- EStack (pile éphémère)
- PStack (pile persistante)

Ainsi que plusieurs notions (au cœur de Sek) :

- Possession $\rightarrow$ Logique de séparation
- Complexité amortie $\rightarrow$ Crédits temps

Sommaire

À propos de CFML

CFML2 est constitué :

- D'une bibliothèque Coq pour la logique de séparation.
- D'un générateur de formules à partir de code OCaml [1].
- D'un ensemble de tactiques pour raisonner avec ces formules.

1. génération de formules caractéristiques en style WP, c'est à dire des WP en logique de séparation pour du code non annoté par ses spécifications.

Les chunks éphémères

On veut définir :

$$\texttt{c} \rightsquigarrow \texttt{EChunk L}$$

Les chunks éphémères

On veut définir :

$$\texttt{c} \rightsquigarrow \texttt{EChunk L} \;==\; (\texttt{EChunk L c} : \texttt{hprop})$$

Les chunks éphémères

On veut définir :

$$\texttt{c} \rightsquigarrow \texttt{EChunk L} \;==\; (\texttt{EChunk L c} : \texttt{hprop})$$

```
Definition EChunk (L:list A) (c:echunk_ A) : hprop :=
  ∃ (t:loc) (s:int) (d:A) (T:list A),
     c ↪ { data' := t; size' := s; default' := d }
   ⋆ t ⇝ Array T
   ⋆ \[EChunk_inv L s d T].
```

- Un echunk est un record mutable → une adresse CFML.

Les chunks éphémères et leur interface

```
Lemma echunk_empty_spec :
  SPEC (echunk_empty d)
    PRE ($(K+1))
    POST (fun c ⇒ c ⇝ EChunk nil).
```

Les chunks éphémères et leur interface

```
Lemma echunk_push_spec :
  ~ (IsFull L) →
  SPEC (echunk_push c x)
    PRE ($1 ⋆ (c ⇝ EChunk L))
    POST (fun _ ⇒ c ⇝ EChunk (x::L)).

Lemma echunk_pop_spec :
  L ≠ nil →
  SPEC (echunk_pop c)
    PRE ($3 ⋆ (c ⇝ EChunk L))
    POST (fun x ⇒ ∃ L', \[L = x::L'] ⋆ (c ⇝ EChunk L')).
```

Les chunks partageables, partagés

Comment faire un nuage...

```
Definition Memory (A:Type)
  : Type :=
  Map.map (echunk_ A) (list A).
```

Les chunks partageables, partagés

Comment faire un nuage... et le ramener sur terre ?

```
Definition Memory (A:Type)
  : Type :=
  Map.map (echunk_ A) (list A).
```

```
Definition Shared (A:Type)
  (M:Memory A) : hprop :=
  Group EChunk M.
```

Les chunks partageables, partagés

Comment faire un nuage... et le ramener sur terre ?

```
Definition Memory (A:Type)
  : Type :=
  Map.map (echunk_ A) (list A).
```

```
Definition Shared (A:Type)
  (M:Memory A) : hprop :=
  Group EChunk M.
```

```
Definition SChunkShared (A:Type)
  (M: Memory A) (L: list A) (s: schunk_ A): Prop :=
  s.support' ∈ dom M
  ∧ SChunk_inv (M[s.support']) L (s.view_size').
```

- `schunk` est un record OCaml immutable → vrai record Coq.
- `SChunkShared` vit dans `Prop` → information duplicable.

Les chunks partagés, et leur interface

```
Lemma schunk_push_spec :
  SChunkShared M L s →
  ∼ (IsFull L) →
  SPEC (schunk_push s x)
    PRE ($(K+3) ⋆ Shared M)
    POST (fun s' ⇒ ∃ M', \[Extend M M'] ⋆ Shared M'
          ⋆ \[SChunkShared M' (x::L) s']).
```

Les chunks partagés, et leur interface

```
Lemma schunk_push_spec :
  SChunkShared M L s →
  ∼ (IsFull L) →
  SPEC (schunk_push s x)
    PRE ($(K+3) ⋆ Shared M)
    POST (fun s' ⇒ ∃ M', \[Extend M M'] ⋆ Shared M'
          ⋆ \[SChunkShared M' (x::L) s']).
```

- `Extend` garantit que les modifications ont été monotones.

Les chunks partagés, et leur interface

```
Lemma schunk_push_spec :
  SChunkShared M L s →
  ∼ (IsFull L) →
  SPEC (schunk_push s x)
    PRE ($(K+3) ⋆ Shared M)
    POST (fun s' ⇒ ∃ M', \[Extend M M'] ⋆ Shared M'
          ⋆ \[SChunkShared M' (x::L) s']).
```

- `Extend` garantit que les modifications ont été monotones.

```
Lemma SChunkShared_mon :
  Extend M M' →
  SChunkShared M L s → SChunkShared M' L s.
```

Une notation pour la monotonie

```
Lemma schunk_push_spec :
  SChunkShared M L s →
  ~ (IsFull L) →
  SPEC (schunk_push s x)
    MONO M
    PRE ($(K+3))
    POST (fun M' s' ⇒ \[SChunkShared M' (x::L) s']).
```

Les piles éphémères

On veut définir :

$$\texttt{e} \rightsquigarrow \texttt{EStack M L}$$

Les piles éphémères

On veut définir :

$$\texttt{e} \rightsquigarrow \texttt{EStack M L} \;==\; (\texttt{EStack M L e} : \texttt{hprop})$$

Les piles éphémères

On veut définir :

$$\texttt{e} \rightsquigarrow \texttt{EStack M L} \;==\; (\texttt{EStack M L e} : \texttt{hprop})$$

```
Definition EStack A
  (M:Memory A) (L:list A) (e:estack_ A) : hprop :=
  ∃ f t id LF LS,
     e ↪ {front' := f; tail' := t; id' := id}
   ⋆ f ⇝ EChunk LF
   ⋆ t ⇝ ListOf (SChunkMaybeOwned M id) LS
   ⋆ \[EStack_inv t id L LF LS]
   ⋆ $(potential (length LF) t id).
```

Des conversions monotones

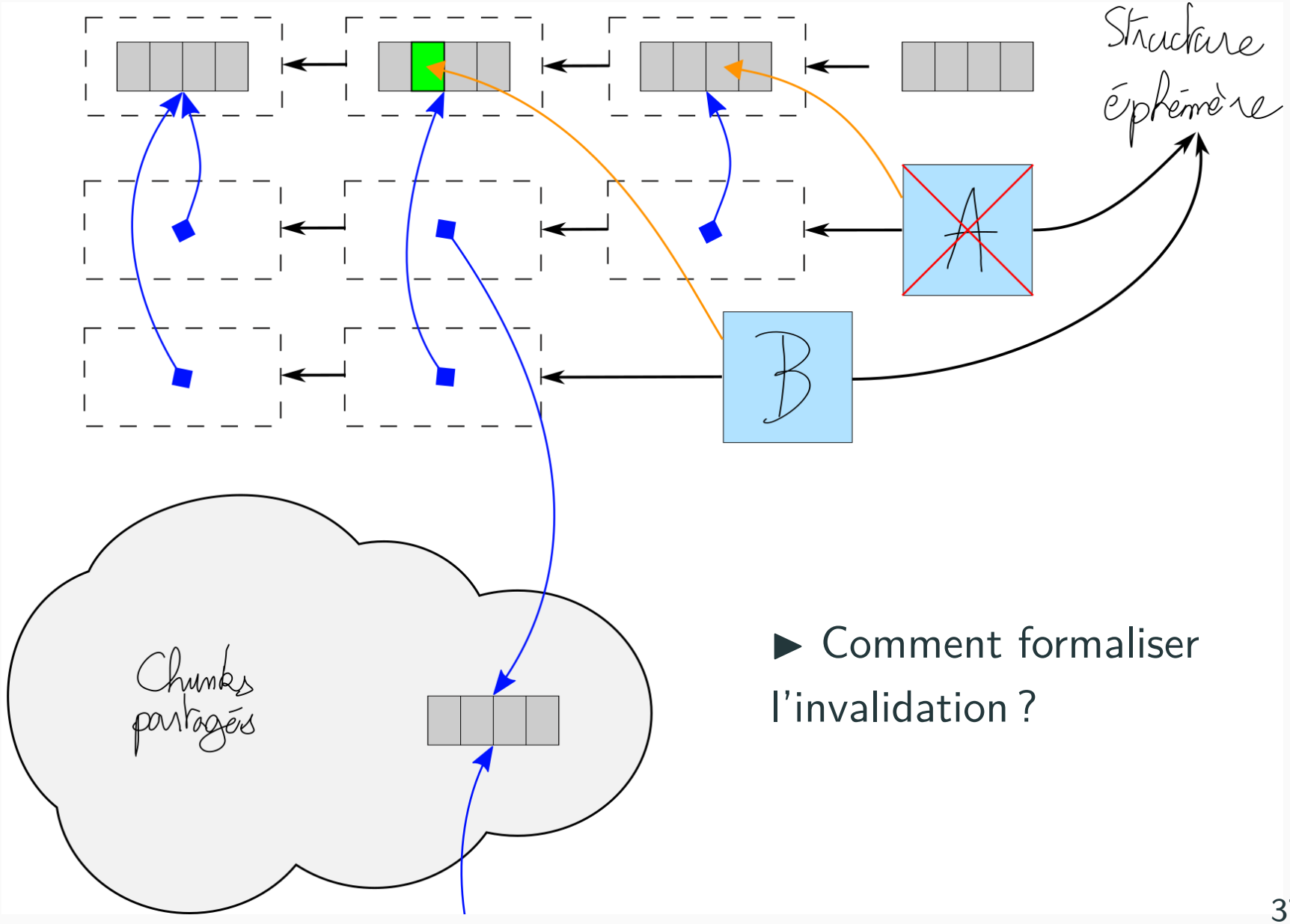
```
Lemma ephemeral_to_persistent_spec :
  SPEC (ephemeral_to_persistent e)
    MONO M
    PRE ($2 ⋆ e ⇝ EStack M L)
    POST (fun M' p ⇒ \[PStack M' L p]).

Lemma persistent_to_ephemeral_spec :
  PStack M L p →
  SPEC (persistent_to_ephemeral p)
    PRE ($(2*K + 4) ⋆ Shared M)
    POST (fun e ⇒ Shared M ⋆ e ⇝ EStack M L).
```

Les itérateurs, pas si compliqué ?

Une spécification simple pour les itérateurs sur pile persistante.

```
Lemma iter_on_pstack_spec :
  PStack M L p →
  SPEC (iter_on_pstack p)
    PRE ($2)
    POST (fun it ⇒ it ⇝ PIterator M L).

Lemma move_spec :
  L ≠ nil →
  SPEC (move it)
    PRE ($4 ⋆ it ⇝ PIterator M L)
    POST (fun _ ⇒ ∃ x L', \[L = x::L'] ⋆ it ⇝ PIterator M L').
```

`set`, rappel du problème



- Comment formaliser l'invalidation ?

Les piles dans tous leurs états

On introduit un prédicat exposant l'état interne de la pile.

```
Lemma EStack_eq :
  e ⇝ EStack M L = ∃ st, e ⇝ EStackInState st M L.
```

- Empêche toute modification de la pile.

Les piles dans tous leurs états

On introduit un prédicat exposant l'état interne de la pile.

```
Lemma EStack_eq :
  e ⇝ EStack M L = ∃ st, e ⇝ EStackInState st M L.
```

- Empêche toute modification de la pile.
- Permet une synchronisation logique.

```
it ⇝ Iterator st i
```

Les itérateurs dans tous leurs états

Les itérateurs sont synchronisés avec leur pile sous-jacente.

```
Lemma get_spec : i ≠ length L →
  SPEC (get it)
    PRE ($3)
    INV (Shared M ⋆ it ⇝ Iterator st i
        ⋆ e ⇝ EStackInState st M L)
    POST (fun x ⇒ x = L[i]).

Lemma move_spec : i ≠ length L →
  SPEC (move it)
    PRE ($4 ⋆ it ⇝ Iterator st i)
    INV (e ⇝ EStackInState st M L)
    POST (fun _ ⇒ it ⇝ Iterator st (i+1)).
```

La spécification invalidante de `set`

```
Lemma set_spec_sharing :
  i ≠ length L →
  SPEC (set it x)
    PRE ($(length L + K + 10) ⋆ it ⇝ Iterator st i
          ⋆ e ⇝ EStackInState st M L)
    INV (Shared M)
    POST (fun _ ⇒ ∃ st', it ⇝ Iterator st' i
                    ⋆ e ⇝ EStackInState st' M (L[i:=x])).
```

- L'état interne change
  → invalidation des itérateurs concurrents.

La spécification conservatrice de `set`

```
Lemma set_spec_no_sharing :
  M = ∅ →
  i ≠ length L →
  SPEC (set it x)
    PRE ($6 ⋆ it ⇝ Iterator st i
          ⋆ e ⇝ EStackInState st M L)
    INV (Shared M)
    POST (fun _ ⇒ it ⇝ Iterator st i
                  ⋆ e ⇝ EStackInState st M (L[i:=x])).
```

- Il s'agit d'une sous-approximation.
- L'état interne est préservé.

Sommaire

Conclusion

Avec CFML :

1. on est capable de prouver un code efficace.
2. on est capable de prouver qu'un code est efficace.

Quelques statistiques :

- $\approx$ 250 lignes d'OCaml.
- $\approx$ 2500 lignes de preuves Coq.
- $\approx$ 1500 lignes de non-preuve (définitions, spécifications...).

Travaux futurs

- Ajouter d'autres fonctions : `iter-segment`, `blit`...
- Comprendre si l'état monotone peut être caché avec Iris [1].
- Combiner ce travail avec la formalisation des arbres de chunks boostrappés, pour ajouter `split` et `concat`.
- (Un jour) Étendre ce travail à Sek, avec toutes les fonctions utilitaires en plus (`filter`, `iter2`...).

1. Jung, Krebbers, Jourdan, Bizjak, Birkedal, and Dreyer (2018)

Merci de votre attention !

À propos d'extension

```
Definition Extend A B (M M':map A (list B)) : Prop :=
  ∀ (i:A),
    i ∈ dom M →
    (i ∈ dom M') ∧ Suffix M[i] M'[i].
```

Bibliographie i

Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Theory and Practice of Chunked Sequences. In Schulz, Andreas S., Wagner, and Dorothea, editors, European Symposium on Algorithms, volume 8737 of Lecture Notes in Computer Science, pages 25 – 36, Wroclaw, Poland, September 2014. Springer Berlin Heidelberg. doi : 10.1007/978-3-662-44777-2_3. URL https://doi.org/10.1007/978-3-662-44777-2_3.

Bibliographie ii

Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming, ICFP '11, page 418–430, New York, NY, USA, 2011. Association for Computing Machinery. ISBN 9781450308656. doi : 10.1145/2034773.2034828. URL https://doi.org/10.1145/2034773.2034828.

Arthur Charguéraud and François Pottier. Verifying the Correctness and Amortized Complexity of a Union-Find Implementation in Separation Logic with Time Credits. Journal of Automated Reasoning, 62(3) :331–365, March 2019. doi :

10.1007/s10817-017-9431-7. URL https://doi.org/10.1007/s10817-017-9431-7.

Arthur Charguéraud and François Pottier. Sek. URL https://gitlab.inria.fr/fpottier/sek/.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Aleš Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up : A modular foundation for higher-order concurrent separation logic. Journal of Functional Programming, 28(e20), 2018. doi : 10.1017/S0956796818000151. URL https://hal.archives-ouvertes.fr/hal-01945446.