Correct tout seul, sûr à plusieurs

Clément Allain et Gabriel Scherer

INRIA

Le modèle mémoire concurrent de OCaml 5 est conçu pour garantir la sûreté du typage et la sûreté mémoire *même* en présence de courses critiques (*data races*) entre fils d'exécution. Ce choix de conception impose des choix techniques précis dans l'implémentation du compilateur et de l'environnement d'exécution du langage. Mais il impose aussi, et c'est le sujet du présent article, des changements dans la façon d'écrire les *programmes utilisateurs* qui utilisent des constructions non sûres comme `Array.unsafe_get` et `Array.unsafe_set` pour des raisons de performance. Même si une structure de données est conçue pour être utilisée de façon séquentielle uniquement, le documente et ne fournit aucune garantie sur un usage concurrent incorrect, elle doit faire les bons choix d'implémentation pour préserver la sûreté mémoire même dans ce cas d'usage incorrect. Des implémentations parfaitement correctes pour OCaml 4 deviennent invalides en OCaml 5.

Nous présentons ici une partie de l'implémentation du module `Dynarray` de tableaux redimensionnables qui a été intégré à la bibliothèque standard de OCaml 5.2. Nous expliquons comment plusieurs choix d'implémentation habituels chez les utilisateurs experts du langage deviennent invalides dans un contexte concurrent. Nous présentons une façon de raisonner sur ces implémentations (≪correct tout seul, sûr à plusieurs≫), comme un modèle mental informel que nous avons utilisé pour écrire ce code, mais aussi dans une formalisation Coq utilisant la logique de séparation Iris. Nous avons développé et vérifié le noyau de `Dynarray` en Iris, et montrons comment organiser les spécifications pour exprimer et vérifier de façon simple et claire cette nouvelle propriété de correction.

1 Introduction

Le drame Un tableau OCaml (`'a array`) a une taille fixe qui ne change pas pendant l'exécution du programme. On a souvent besoin d'un tableau redimensionnable, auquel il est possible d'ajouter ou d'enlever des éléments.

La représentation classique d'un tableau redimensionnable dont la taille est actuellement n utilise un enregistrement contenant deux champs modifiables :

— Un *tableau support* `arr : 'a array`, de taille au moins n, contenant tous les éléments du tableau redimensionnable et un peu d'espace libre pour ajouter de nouveaux éléments.

— Un entier `length` ou `position` qui stocke la valeur actuelle de n ou, de façon équivalente, la position où il faudra écrire le prochain élément ajouté au tableau.

```
type 'a dynarray = {
  mutable arr: 'a array;
  mutable length: int;
}
```

Nous appelons *capacité* la taille du support, qui doit être au moins égale à la *taille* actuelle du tableau redimensionnable.

```
let capacity (a: 'a dynarray) : int = Array.length a.arr
let length  (a: 'a dynarray) : int = a.length
```

Ajouter des élément utilise une fonction `ensure_capacity` : un appel à `ensure_capacity a k` s'assure que la capacité de `a` est au moins égale à `k`. Si cette condition n'est pas vérifiée, la fonction fait une modification en place en remplaçant `arr` par un nouveau tableau de taille suffisante – typiquement en multipliant sa taille par une constante, ce qui permet d'obtenir une bonne complexité amortie.

Ajouter un élément en fin de tableau demande la capacité pour `1` élément de plus :

```
let add_last a v =
  let len = a.length in
  ensure_capacity a (len + 1);
  Array.unsafe_set a.arr len v;
  a.length <- len + 1
```

Le code de `add_last` utilise la fonction `Array.unsafe_set` pour écrire directement dans le tableau sans tester que l'indice est dans les bornes, ce qui serait redondant avec les tests déjà effectués dans `ensure_extra_capacity`.

Ce code est parfaitement correct en OCaml 4, et parfaitement faux en OCaml 5. En effet, un utilisateur ou une utilisatrice pourrait utiliser d'autres fonctions qui modifient le tableau `a.arr` en parallèle avec l'appel à `add_last`. Si cette modification est observée entre l'appel à `ensure_capacity` et l'appel à `unsafe_set`, le tableau `a.arr` peut changer et devenir un tableau plus petit – par exemple un appel à `Dynarray.clear` qui met la taille à `0` et libère la mémoire du tableau de support précédent. `unsafe_set` casse alors la sûreté mémoire (et donc de typage) du langage.

C'est frustrant puisque cette structure de données *séquentielle*, pas concurrente, ne doit pas être utilisée en parallèle sans synchronisation. Du code qui ferait de tels accès est buggé, il ne respecte pas la spécification de la bibliothèque `Dynarray`. Mais comme on ne peut pas empêcher ce bug par typage, il faut gérer ce cas en garantissant au moins la sûreté mémoire. Le sujet de cet article n'est pas l'implémentation d'un tableau dynamique concurrent, mais d'un tableau dynamique *séquentiel* qui reste «sûr à plusieurs».

Par exemple, un correctif possible pour `add_last` serait la version suivante :

```
let add_last a v =
  let len = a.length in
  let arr = with_capacity a (len + 1) in
  Array.unsafe_set arr len v;
  a.length <- len + 1
```

avec une fonction `with_capacity a k` qui modifie `a` en place si nécessaire, mais qui en plus renvoie le tableau de support de taille au moins `k` – le tableau en entrée ou le nouveau tableau agrandi. L'écriture non sûre se fait alors sur un tableau dont on sait qu'il est de taille au moins `k`. Cette fonction n'est toujours pas correcte dans un contexte concurrent, mais elle est sûre. (Écrire une version concurrente serait utile dans plus de contextes, mais le code serait aussi sensiblement plus lent car les opérations de synchronisation coûtent plus cher.)

Nos questions Comment écrire des structures de données séquentielles en OCaml 5 qui ne cassent pas les garanties de sûreté dans un cadre concurrent ? Comment relire ce code pour s'assurer qu'il est correct ? Pouvons-nous formaliser précisément la situation pour nous assurer que les réponses aux questions précédentes sont correctes ?

(Question bonus : combien d'implémentations malines et rapides de structures de données OCaml sont complètement non-sûres en OCaml 5 à cause de ce genre de problèmes ?)

Contributions Les contributions présentées dans cet article sont les suivantes :

— Une implémentation du module `Dynarray` intégrée dans la bibliothèque standard OCaml. Nous ne décrivons pas l'ensemble de l'interface (qui vient en grande partie d'un travail amont de Simon Cruanes), mais nous concentrons sur le problème de la sûreté dans un contexte concurrent. Nous parlons aussi, moins en détail, de deux autres problèmes de conception délicats : l'emboîtement (*boxing*) des éléments du tableau et l'invalidation des itérateurs.

— Une formalisation dans la logique de séparation Iris du critère de correction qui a guidé l'écriture du code et sa relecture, ≪correct tout seul, sûr à plusieurs≫, et une implémentation vérifiée d'un petit noyau de `Dynarray` qui établit ce critère de correction.

2 `Dynarray` dans OCaml 5

2.1 Usages

Il existe déjà de nombreuses implémentations OCaml de tableaux dynamiques, comme des bibliothèques séparées ou même cachées dans des projets plus importants. Nos deux implémentations de référence sont Vector [1] de Jean-Christophe Filliâtre et le module CCVector de la bibliothèque containers de Simon Cruanes. À l'heure où nous écrivons ces lignes, au début octobre 2023, les implémentations de Vector et CCVector sont d'ailleurs incorrectes en OCaml 5 : elles utilisent par exemple la version non-sûre de `add_last` que nous avons discutée en introduction.

Un tableau redimensionnable est une structure de données très polyvalente. On peut la voir comme un tableau dont le nombre d'éléments change au cours du temps, mais aussi comme une pile (ou une liste modifiée en place) qui permet en plus un accès en position arbitraire.

Dans certains idiomes de programmation très impératifs, les tableaux dynamiques remplacent les listes (persistantes) comme structure de données de base pour accumuler des éléments, itérer dessus, etc.

Pour des exemples plus spécifiques, c'est par exemple une bonne structure de données pour implémenter la trace (*trail*) d'un prouveur automatique (ou d'autres structures de journal avec *rebroussement* (*backtracking*), pour pouvoir empiler et dépiler efficacement des décisions, mais aussi remonter inspecter des décisions sans les dépiler. On peut aussi s'en servir pour stocker l'ensemble des clauses apprises, etc. Les prouveurs Zenon et Zipperposition, par exemple, font un usage très intensif de tableaux dynamiques.

C'est aussi une structure de données utilisée pour implémenter de nombreux algorithmes, par exemple pour stocker une `heap` (un arbre binaire de recherche stocké dans un tableau dynamique) pour implémenter le tri par tas ou une file de priorité.

2.2 API

Cette section propose un tour rapide de l'API de `Dynarray` telle que proposée dans la bibliothèque standard d'OCaml 5.2. On ne montre pas toutes les fonctions, seulement les exemples représentatifs de chaque classe de fonction. Nous donnons seulement le nom et le type de chaque fonction, laissant au lecteur le loisir d'inférer leur comportement – ou de lire la documentation rédigée avec amour.

Les fonctions de base d'un tableau dynamique sont similaires à celles d'un tableau de taille fixe, par exemple :

```
val create : unit -> 'a t
val make : int -> 'a -> 'a t
```

1. Le nom ≪vector≫ pour les tableaux dynamiques est fameusement utilisé par les bibliothèques standard C++ (`std::vector`) et Rust ().

```
val init : int -> (int -> 'a) -> 'a t

val get : 'a t -> int -> 'a
val set : 'a t -> int -> 'a -> unit

val length : 'a t -> int
```

Il est aussi possible d'utiliser le tableau comme une pile, en ajoutant et enlevant des éléments en fin de tableau, par exemple :

```
val get_last : 'a t -> 'a
val add_last : 'a t -> 'a -> unit
val pop_last_opt : 'a t -> 'a option
```

Cette fonction d'addition en fin de tableau se généralise à des fonctions pour ajouter diverses structures de données – permettant aux tableaux dynamique de jouer le rôle de *buffer* où l'on verse des données de formats variés. Par exemple :

```
val append_array : 'a t -> 'a array -> unit
val append_list : 'a t -> 'a list -> unit
val append : 'a t -> 'a t -> unit
val append_seq : 'a t -> 'a Seq.t -> unit
```

Un ensemble de fonctions assez riche est fourni pour itérer sur les éléments, par exemple :

```
val iteri : (int -> 'a -> unit) -> 'a t -> unit
val mapi : (int -> 'a -> 'b) -> 'a t -> 'b t

val fold_left : ('acc -> 'a -> 'acc) -> 'acc -> 'a t -> 'acc
val fold_right : ('a -> 'acc -> 'acc) -> 'a t -> 'acc -> 'acc

val exists : ('a -> bool) -> 'a t -> bool
val for_all : ('a -> bool) -> 'a t -> bool

val filter : ('a -> bool) -> 'a t -> 'a t
val filter_map : ('a -> 'b option) -> 'a t -> 'b t
```

On peut convertir entre les tableaux dynamiques et différentes structures de données :

```
val of_array : 'a array -> 'a t
val to_array : 'a t -> 'a array

val of_list : 'a list -> 'a t
val to_list : 'a t -> 'a list

val of_seq : 'a Seq.t -> 'a t
val to_seq : 'a t -> 'a Seq.t
```

Enfin, un ensemble de fonctions plus avancées permet de jouer avec la taille du tableau de support, quand un expert veut faire mieux que la politique de redimensionnement par défaut pour un cas d'usage particulier.

```
val capacity : 'a t -> int
val ensure_capacity : 'a t -> int -> unit

val fit_capacity : 'a t -> unit
val set_capacity : 'a t -> int -> unit
```

Nous avons déjà parlé de la fonction `ensure_capacity a n`, qui garantit une capacité d'au moins `n` pour le tableau dynamique `a`. Elle est utile pour éviter les redimensionnements inutiles si on connaît à l'avance la taille finale qu'aura le tableau.

La fonction `fit_capacity` réduit le tableau de support du tableau dynamique pour que sa taille soit exactement la taille nécessaire et pas plus. Elle est utile pour éviter de conserver un espace mémoire supplémentaire si l'on sait statiquement que le tableau ne va plus recevoir de nouveaux éléments. (Notre implémentation ne réduit pas implicitement la taille du tableau de support quand le nombre d'élément diminue, donc ces fonctions pour réduire explicitement sa taille sont parfois utiles.)

La fonction `set_capacity a n` réalloue un tableau de support de capacité exactement `n`. On peut s'en servir comme une version plus précise de `ensure_capacity` pour agrandir le tableau, ou au contraire pour réduire un tableau en supprimant certains éléments. En particulier, `set_capacity a 0` libère tout l'espace mémoire occupé par un tableau dynamique.

2.3 Choix de conception : vivacité mémoire

Un choix de conception est le comportement des fonctions qui retirent des éléments au tableau dynamique. Une implémentation naturelle serait la suivante :

```
let pop_last_opt a =
  let n = a.length in
  if n = 0 then None
  else begin
    let v = a.arr.(n - 1) in
    a.length <- n - 1;
    Some v
  end
```

Mais cette version a le défaut que la valeur `v` en fin de tableau, bien que retirée du tableau tel que vu par l'utilisateur, est toujours accessible dans la mémoire du tableau de support, et sera conservée en vie par le glaneur de cellules.

Notre implémentation garantit une propriété de vivacité mémoire minimale : les valeurs accessibles dans le tableau de support `a.arr` sont exactement les valeurs des éléments aux positions de `0` à `a.length - 1`. En particulier, aucune valeur initialement fournie par l'utilisateur n'est gardée en vie indépendamment par le tableau de support dans l'espace ≪vide≫ après la fin du tableau.

Pour obtenir cette garantie il est nécessaire d'effectuer des écritures supplémentaires pour effacer les valeurs retirées du tableau. La question ≪avec quoi écraser ces valeurs ?≫ nous amène à un deuxième choix de conception un peu délicat.

2.4 Choix de conception : la valeur du vide

Si le tableau de support d'un `'a dynarray` est de type `'a array`, pour que `pop_last_opt` préserve une vivacité minimale il faut ≪effacer≫ le dernier élément du tableau en y mettant une autre valeur de type `'a`. Quelle valeur choisir pour le vide ?

2.4.1 Un des éléments visibles

On peut utiliser un des éléments du tableau, par exemple celui en position `0` – à condition de remplacer le tableau de support par un tableau vide quand il n'a plus d'éléments. Mais ce choix demande de parcourir tout le tableau de support à chaque fois que l'élément en position `0` est modifié, donnant une mauvaise surprise pour la complexité de `set`.

2.4.2 Un élément fourni explicitement

On peut demander à l'utilisateur de fournir une valeur ≪par défaut≫ au moment de la création du tableau, stockée comme un troisième champ de l'enregistrement, qui est documentée comme restant vivante pendant toute la durée de vie du tableau. On relâche

un peu la promesse de vivacité minimale, mais de façon explicite et contrôlée. C'est le choix de la bibliothèque Vector. Nous n'avons pas fait ce choix, car il alourdit l'API, toutes les fonctions de création doivent demander une valeur supplémentaire. Si on veut implémenter `map : ('a -> 'b) -> 'a dynarray -> 'b dynarray` par exemple, faut-il dériver un élément par défaut dans `'b` en appelant `f` sur l'élément par défaut dans `'a`, ou demander un nouvel élément par défaut ? Enfin, certains usages de `Dynarray` impliquent des types d'éléments complexes. Il peut être pénible pour l'utilisateur de construire un élément par défaut.

2.4.3 Une valeur invalide

L'implémentation proposée par Simon Cruanes utilisait une valeur non-sûre de la forme `(Obj.magic () : 'a)` comme élément par défaut. [2]

Le problème de cette approche est qu'elle est incorrecte en OCaml 5. Considérons une implémentation courante de `get` :

```
let get a i =
  if i >= a.length then invalid_arg "Dynarray.get";
  a.arr.(i)
```

En présence d'accès concurrents imprévus, il n'y a aucune garantie que la taille `a.length` soit inchangée au moment où on accède à `a.arr` ; un appel concurrent à `pop_last_opt` pourrait avoir modifié `a.length` et mis un élément par défaut en position `i`, qui peut être alors renvoyé par la fonction `get`. Si l'on utilise un élément par défaut qui n'est pas une valeur valide à ce type, on peut casser la sûreté du typage et donc la sûreté mémoire. [3]

Une façon de contourner ce problème serait de tester, dans `get`, si la valeur lue est la valeur par défaut, et d'échouer dans ce cas.

```
let get a i =
  if i >= a.length then invalid_arg "Dynarray.get";
  let v = a.arr.(i) in
  if is_dummy_value a v then invalid_arg "Dynarray.get";
  v
```

Ce n'est pas possible si l'on utilise `()` comme valeur par défaut, puisque cette valeur pourrait avoir été insérée par l'utilisateur. Quand nous avons fait remonter ce problème de concurrence avec l'implémentation de Simon Cruanes, une longue discussion a fait émerger deux choix possibles :

1. On peut utiliser une valeur allouée dynamiquement et jamais montrée à l'utilisateur, par exemple `ref ()`, stockée comme une constante (privée) du module ou dans un champ du tableau.
2. On pourrait aussi utiliser des valeurs magiques permises par le runtime mais qui ne représentent aucune valeur OCaml source valide.

2.4.4 Représentation emboîtante

La solution la plus simple, et celle que nous avons finalement adoptée dans notre proposition d'implémentation, est d'utiliser un type différent de `'a array` pour le tableau de support. Si on utilise `'a option array`, il suffit de mettre `None` dans les cases non utilisées du tableau de support.

Il est en fait possible d'utiliser une légère variante du type `'a option` avec de meilleures propriétés pour cet usage. C'est ce que nous faisons dans notre implémentation :

2. Et pour les nombres flottants ? Il faut complexifier un peu le code pour les gérer de façon sûre, en raison des optimisations de représentation des tableaux de flottants.

3. Par contraste, avec les approches précédentes, on peut renvoyer une "valeur vide" dans ce cas. Les valeurs utilisées pour les cases vides sont valides au type `'a` donc il n'y a pas de problème de sûreté.

```
type 'a slot =
| Empty
| Elem of { mutable v: 'a }
```

L'intérêt d'utiliser un `'a slot array` plutôt qu'un `'a option array` est qu'il n'y a pas besoin d'allouer pour implémenter `set` :

```
(* version avec [arr : 'a option array] *)
let set a i v =
  a.arr.(i) <- Some v (* allocation *)

(* version avec [arr : 'a slot array] *)
let set a i v =
  match a.arr.(i) with
  | Empty -> invalid_arg "Dynarray.set"
  | Elem e -> e.v <- v (* modification en place *)
```

Il faut toujours allouer ce constructeur `Elem` quand on ajoute de nouveaux éléments au tableau, mais (contrairement à `'a option`) pas quand on s'en sert comme un tableau de taille fixe. (La modification en place a aussi un coût, supérieur parfois au coût de l'allocation, mais il y a de toute façon une écriture dans les deux versions.)

Nos utilisateurs potentiels étaient très inquiets de l'impact sur les performances de cette représentation *emboîtée* par rapport à `'a array`. En particulier, cette représentation est moins compacte en mémoire – elle utilise un mot de plus par élément et l'accès à un élément contient une indirection, ce qui pourrait coûter cher sur des cas d'usages intensifs brassant beaucoup de mémoire en raison d'une moins bonne localité mémoire. Ceci dit, le constructeur `Elem` est souvent alloué en même temps ou presque que la valeur qu'on ajoute au tableau, et les GC générationnels sont bons pour transformer la localité temporelle en localité spatiale.

Selon nos mesures, on observe un ralentissement finalement assez faible. Nous avons fait l'expérience de modifier un SAT-solver, qui faisait un usage critique de tableaux dynamiques, pour utiliser cette représentation moins efficace. Dans la plupart des cas les performances n'ont pas vraiment changé, nous avons trouvé des ralentissements de 10-15% pour certains fichiers de test, et 25% sur un seul cas de test. [4]

2.5 Choix de conception : invalidation des itérateurs

Le dernier choix de conception qui a fait couler beaucoup d'encre pour `Dynarray` est l'invalidation des itérateurs. Comment réagir si la taille du tableau est modifiée *pendant* qu'une fonction est en train d'itérer sur le tableau (par exemple `iter`, `map`, `exists`, mais aussi `to_list` ou `to_seq`) ? Une telle modification peut venir d'un usage concurrent de la structure, mais aussi d'une situation de réentrance – par exemple si l'itérateur appelle une fonction utilisateur sur chaque élément, qui ajoute ou retire des éléments au tableau.

Plusieurs choix s'offrent à nous, par exemple :

— Donner le droit à la fonction d'itération d'observer ou non chaque modification. Une implémentation conforme pourrait prendre en compte certains ajouts et suppressions mais aussi continuer à ≪voir≫ des éléments supprimés ou ≪manquer≫ des éléments ajoutés.

— Imposer une sémantique spécifique dans ce cas, par exemple demander à ce que l'itération observe toutes les modifications, ou au contraire soit équivalente à une itération sur le tableau tel qu'il était au départ.

— Donner le droit à la fonction d'itération d'échouer si elle observe une modification indépendante.

— Imposer que la fonction d'itération échoue si elle observe une modification indépendante.

4. https://github.com/ocaml/ocaml/pull/11882#issuecomment-1405867624

(La notion de «modification» ici est une modification de structure du tableau, c'est-à-dire de sa taille, en ajoutant ou retirant des éléments. Il y a aussi les modifications d'éléments (sans changer de structure), obtenues avec `Dynarray.set`. On peut imaginer de faire des choix différents ci-dessus pour les modifications de structure et les modifications d'éléments.)

Permettre plus de comportements a l'avantage de permettre des implémentations plus efficaces, mais augmente les chances que l'utilisateur ou utilisatrice introduise des bugs subtils liés à des comportements permis mais non observés pendant ses tests. À l'inverse, imposer un comportement précis simplifie la programmation mais peut être plus coûteux.

Une fois la discussion sur le boxing tassée, une très large part de la discussion de notre travail sur `Dynarray` a été consacrée à ces questions d'invalidation des itérateurs. Nous avons soumis plusieurs implémentations faisant des choix distincts, par exemple une implémentation maximisant les performances avec une spécification très permissive, ou une implémentation qui observe toutes les modifications (de structure et d'éléments) dans le cadre séquentiel sans échouer.

Le consensus auquel la discussion a abouti correspond à une position poussée par Simon Cruanes et Guillaume Munch-Maccagnoni :

— À ce jour il n'y a pas de cas d'usage clairement identifié pour l'usage d'itérateurs en même temps que des modifications de structure, et cette pratique cache habituellement une erreur de programmation.
— Il vaut donc mieux interdire les modifications de structure pendant l'itération, et échouer systématiquement.
— Un utilisateur ou une utilisatrice souhaitant prendre en compte des modifications de structure pendant l'itération peut implémenter ses propres itérateurs, et/ou devrait utiliser une structure de données conçue dès le départ pour un usage concurrent.

La spécification choisie est la suivante : effectuer une modification de structure pendant une itération est une erreur de programmation, peut lever une exception `Invalid_argument`, et l'implémentation fera des efforts raisonnables pour détecter ces cas et échouer.

Les itérateurs de notre implémentation échouent s'ils atteignent un élément qui a été retiré du tableau après le début de l'itération, et vérifient en fin d'itération que la taille du tableau n'a pas changé – donc ils échouent si des éléments ont été rajoutés. Le coût de ces vérifications est très faible, négligeable dans du code utilisateur typique. Cette approche peut manquer des modifications (elle peut échouer à échouer), par exemple si la fonction utilisateur ajoute temporairement des éléments avant d'en retirer le même nombre, ou en général effectue des modifications de structure qui préservent le nombre d'éléments. Voici notre implémentation de `Dynarray.iteri` :

```
let iteri k a =
  let {arr; length} = a in
  check_valid_length arr length;
  for i = 0 to length - 1 do
    k i (unsafe_get arr i length);
  done;
  check_same_length a length
```

La fonction `check_valid_length` vérifie l'invariant `length <= Array.length arr`, ce qui permet d'éliminer les tests d'accès au tableau support dans le corps de la boucle. La fonction `unsafe_get` fait un accès non sûr au tableau support, mais échoue sur des éléments vides (qui ont été retirés du tableau). Notons que le corps de la boucle utilise `arr` et `length`, le tableau support et la taille au début de la boucle, sans refaire de lecture de champs modifiables qui pourraient observer des modifications de structure concurrentes. Enfin, `check_same_length` vérifie en fin de boucle que la taille du tableau est restée celle de départ. C'est un test de confort ou de correction, qui sert à échouer plus souvent en cas de comportement invalide.

2.5.1 Aller plus loin ?

On pourrait imaginer une implémentation qui échouerait dans tous les cas de modification de structure, en stockant un ≪numéro de version≫ incrémenté à chaque modification de structure. Pour une implémentation correcte dans le cadre concurrent il faudrait un numéro de version atomique, et ajouter un incrément atomique à chaque `Dynarray.add_last` aurait un coût significatif. Rendre ce champ non-atomique diminue le coût en performance mais rend l'approche incomplète de nouveau. Cette piste d'amélioration, qui est utilisée dans le module `ArrayList` bibliothèque standard Java, a été suggérée par Guillaume Munch-Maccagnoni. Nous ne l'avons pas encore explorée.

Remarque 1. Au lieu d'écrire à chaque modification et de lire dans les itérateurs, on peut imaginer une version où les itérateurs modifient la structure pour indiquer qu'une itération est en cours (en mettant un drapeau atomique à `true`), et les modifications testent cette situation et échouent dans ce cas.

Cette approche serait plus efficace mais elle a le défaut majeur qu'elle fait des itérateurs des opérations qui modifient la structure de données. En particulier, dans le cadre d'un programme concurrent, un utilisateur pourrait vouloir faire deux itérations en parallèle sans synchronisation, tant que les itérations ne font que lire le tableau. (Avoir une possession unique pendant des phases d'écriture, mais une possession partagée pendant des phases de lecture seule.) Ce cas d'usage est tout à fait valide avec les implémentations habituelles, mais devient invalide si les itérateurs modifient l'état en place. Autrement dit, dans un cadre concurrent ajouter des modifications en place peut changer la spécification d'une fonction, et faire d'un itérateur une fonction modifiante est une très mauvaise idée.

(Le module `Hashtbl` de la bibliothèque standard utilise depuis 2015 des modifications en place pendant l'itération, pour des raisons de performance, et a ce problème.)

2.5.2 Itérateurs externes, générateurs.

Un cas particulier d'itérateur est

```
val to_seq : 'a dynarray -> 'a Seq.t
```

qui est un itérateur ≪externe≫ : une séquence de type `'a Seq.t` est énumérée à la demande. Dans un programme séquentiel, il est rare qu'un énumérateur strict comme

```
val to_list : 'a dynarray -> 'a list
```

puisse observer une invalidation – c'est possible en cas d'appels asynchrones (finaliseurs, etc.) causé par une allocation. C'est plus facile pendant l'exécution de

```
val iter : ('a -> unit) -> 'a dynarray -> unit
```

puisque la fonction fournie par l'utilisateur peut modifier le tableau dynamique. Dans le cas de `to_seq`, l'invalidation peut venir après que la fonction a renvoyé une valeur, quand le contexte d'appel forcera cette séquence.

Nous avons convergé vers une API proposant deux versions de `to_seq` : la version de base, nommée `to_seq`, échoue comme les autres itérateurs, si l'accès à un élément de la séquence a lieu après modification du tableau dynamique. Une variante `to_seq_reentrant` n'échoue pas dans ce cas : accéder au i-ème élément de la séquence générée renvoie le i-ème élément du tableau dynamique au moment de l'accès, ou `Empty` si le tableau s'arrête avant.

2.6 Mesures de performance ?

Mesurer précisément les performances d'une telle structure de données est fastidieux et très difficile à la fois. Pour être tout à fait honnêtes, nous ne l'avons pas bien fait. Nous rencontrons au moins les difficultés suivantes :

— Les microbenchmarks magnifient les différences de performance entre opérations très rapides, qui constituent une fraction très faible du temps total de calcul dans la plupart des applications finales.

— Au contraire, les microbenchmarks échouent typiquement à mesurer les effets de localité mémoire qui s'observent sur des programmes qui, à la fois, font un usage intense de gros tableaux dynamiques, et manipulent de gros volumes de données par ailleurs.

Pour donner une idée d'ordres de grandeur sur le premier point, voici les résultats d'un microbenchmark des opérations `get` et `set` sur une structure à quelques milliers d'éléments :

— `Dynarray` est environ 2x plus lent que `Array` (un test d'indice en plus à chaque accès).
— `Hashtbl` est environ 42x plus lent que `Array` (un calcul d'empreinte à chaque accès)
— `Map` est environ 78x plus lent que `Array` (un parcours d'arbre équilibré à chaque accès)

Les opérations déterminantes pour `Dynarray` nous semblent être `get` et `set`, d'une part, et `add_last` et `pop_last` d'autre part. Les autres opérations sont moins fréquentes et rarement dans les parties critiques pour les performances d'un programme.

Une fois ces pincettes prises, voici les ordres de grandeurs tirés de microbenchmarks effectués pendant la préparation de cette PR, en décembre-janvier 2023 :

— Emboîter les éléments n'apporte pas de surcoût mesurable sur `get` et `set`.
— Emboîter les éléments apporte un petit surcoût sur des séquences de `add_last` seules, qui peuvent apparaître dans des cas de construction itérative d'une collection finale : 1.6x plus lent sur des cas utilisant `ensure_capacity` pour limiter les redimensionnements, et 2x sinon.
(Dans le cas particulier de `append_array` on observe des différences plus marquées de 3x-4x, parce qu'utiliser la représentation déboîtée `'a array` permet d'implémenter `append_array` en utilisant `Array.blit` directement.)
— Les différences de performance liées aux différents choix de conception pour l'invalidation des itérateurs sont très faibles. Entre notre version optimisée pour les performances et notre version optimisée pour échouer le moins souvent, les différences étaient de l'ordre de 1.2-1.3x dans certains cas, 1.6x dans les cas les plus favorables à la version optimisée pour l'efficacité.

Globalement, nos tests suggèrent que les implémentations de la seconde PR de Gabriel Scherer ont des performances comparables à celle de la première PR par Simon Cruanes, malgré des évolutions notables sur les choix de spécification et d'implémentation.

2.7 Histoire de la Pull Request (PR) pour OCaml 5

Plusieurs bibliothèques de tableaux dynamiques existent en OCaml, mais une structure de données devient encore plus facile d'accès et plus utilisée quand elle intègre la bibliothèque standard. Le projet `extlib`, qui voulait proposer une extension communautaire largement utilisée à la bibliothèque standard minimale, avait déjà une implémentation de `Dynarray`, qui semble provenir de code écrit par Brian Hurt en 2003 ; ce code a été hérité par le projet `Batteries`, lui-même une inspiration de la bibliothèque `containers` de Simon Cruanes contenant `CCVector`.

Faire entrer du code dans la bibliothèque standard OCaml est cependant une tâche difficile : en cas d'hésitation sur la réponse à une question de conception, l'équipe qui la maintient décide souvent de bloquer les changements proposés plutôt que de trancher de façon arbitraire. Ajouter une ou deux fonctions est difficile, ajouter un nouveau module relève, en toute modestie, du tour de force.

2.7.1 Première PR par Simon Cruanes

En avril 2022, Simon Cruanes a décidé de tenter l'impossible et de faire entrer un module de tableaux extensibles dans la bibliothèque standard. Une réunion de discussion avec Florian Angeletti et Gabriel Scherer a permis de dégager une proposition minimale [5]. Des trois problèmes de conception que nous avons mentionné (concurrence, valeurs vides, invalidation des itérateurs), nous avions seulement identifié celui des valeurs vides, pour lequel Simon

5. MVP, Mise-en-oeuvre Vaguement Potable ?

Cruanes proposait une approche à la `Obj.magic ()`, éventuellement implémentée en C pour éviter toute inquiétude sur le cas des tableaux de flottants. La proposition contenait une API assez courte, inspirée de `CCVector` et aussi notablement des choix de conception de la bibliothèque de Rust. Simon Cruanes était désigné volontaire pour l'implémenter.

De façon indépendante, en mai 2022 les tests aléatoires de Jan Midtgaard et l'analyse détaillée de Jon Ludlam ont découvert une erreur de concurrence dans le code de `Buffer`, rapportée par David Allsopp en #11279 et corrigée par Florian Angeletti en #11742. On peut voir `Buffer` comme une version simplifiée de `Dynarray`, spécialisée au type `char` et sans accès à une position arbitraire. C'est le moment où les développeurs de la bibliothèque standard se sont rendu compte des problèmes d'interaction entre concurrence et accès non-sûr dans du code existant. Notons que la correction du problème n'a pas dégradé les performances – c'est même le contraire, l'attention portée aux micro-optimisations pendant sa correction a permis d'améliorer légèrement les performances de `Buffer`.

En septembre 2022, Simon Cruanes envoie sa proposition d'implémentation de tableaux dynamiques, #11563. L'implémentation de départ utilise `Obj.magic` pour créer des valeurs vides, mais Simon Cruanes identifie le problème de la concurrence et change l'interface et l'implémentation pour mélanger des arguments explicites (dans `ensure_capacity`) et l'usage des éléments existants du tableau, abandonnant la propriété de vivacité minimale. Une longue discussion (92 messages) s'ensuit en septembre et octobre 2022, avec en particulier un débat sur la valeur du vide à choisir (valeur privée ou valeur invalide ?), et une alerte de Guillaume Munch-Maccagnoni sur le problème d'invalidation des itérateurs. La valeur du vide divise les participants et la situation semble bloquée.

2.7.2 Deuxième PR par Gabriel Scherer

Gabriel Scherer propose en janvier 2023 une nouvelle implémentation de `Dynarray` par-dessus le travail de Simon Cruanes, modifiée pour «emboîter» les éléments dans un type option modifiable que nous avons décrit en Section 2.4.4. Le premier but est de convaincre les participants qu'une représentation emboîtante, bien que légèrement moins efficace, peut permettre d'avancer en remettant à plus tard le choix d'une approche non-sûre optimisée. Ceci est permis par des mesures de performances qui indiquent un coût relativement faible pour l'emboîtement.

Cette PR pointe aussi du doigt le problème d'invalidation des itérateurs. Elle n'indique pas de préférence entre les possibilités évoquées en Section 2.5, et propose deux alternatives, qui n'échouent pas en cas de modification concurrentes. La première version permet à l'implémentation d'observer ou non les modifications, maximisant les performances, l'autre impose une spécification simple et plutôt naturelle (chaque accès opère sur la version la plus récente du tableau observée par le fil d'exécution), légèrement moins efficace.

La version actuelle de la PR contient 46 fonctions, une interface de 641 lignes (surtout des commentaires) et une implémentation de 791 lignes. Un effort particulier est fait sur la qualité de la documentation du module, et sur des messages d'erreur les plus clairs et informatifs possibles.

La très longue discussion (337 messages) peut se résumer par les grands moments suivants :

— Janvier 2023 : on atteint un consensus sur le fait qu'une implémentation emboîtante est acceptable comme un premier pas.

— Janvier 2023 : Guillaume Munch-Maccagnoni et Simon Cruanes insistent pour obtenir une spécification où l'invalidation des itérateurs échoue et est considérée comme une erreur de programmation, au lieu que le code s'accommode de ces situations sans échouer. Une troisième version de l'implémentation est nécessaire.

— Mars 2023 : Daniel Bünzli relit l'interface et sa documentation, qui évolue nettement en réponse à ses commentaires.

— Avril 2023 : Clément Allain relit l'implémentation pour vérifier (informellement) sa correction. C'est de cette interaction qu'est née le présent article. C'est Armaël Guéneau qui a mis le doigt sur l'idée de raisonner avec deux spécifications superposées : une

spécification forte et précise dans le cas séquentiel, cherchant à établir la correction fonctionnelle ; une spécification plus faible dans le cas concurrent, suffisant à garantir la sûreté mémoire.
— Mai 2023 : relecture d'ensemble par Damien Doligez qui «approuve» la PR. (Pour les modifications à la bibliothèque standard, les règles de développement OCaml demandent que deux mainteneurs différents approuvent chaque changement.)
— Juin 2023 : relecture de la documentation et des messages d'erreur par Wiktor Kuchta.
— Juillet 2023 : relecture des questions de concurrence, réentrance et du choix des exceptions par Guillaume Munch-Maccagnoni.
— Mai et juillet : discussions sur `to_seq` (le cas des itérateurs externes, voir Section 2.5.2), consensus final sur une version avec une variante réentrante explicite.
— Septembre 2023 : relecture d'ensemble par Florian Angeletti qui «approuve» la PR, obtenant le quota de deux approbation de mainteneurs.

3 Raisonner sur `Dynarray`

L'examen de la PR décrite précédemment nous a conduit à adopter et vérifier (d'abord informellement) les critères de correction suivants :

1) Correction fonctionnelle. L'implémentation des fonctions de `Dynarray` respecte la spécification (informelle) donnée dans l'API. Autrement dit, chaque fonction se comporte comme attendu *si le programmeur l'utilise lui-même de façon correcte* – purement séquentielle ou concurrente mais synchronisée. Le comportement est indéfini si le programmeur en fait un usage concurrent non synchronisé.

Pour le vérifier, on raisonne dans un *cadre séquentiel* en s'appuyant sur un *invariant fort* affirmant notamment : à tout moment, si ce n'est transitoirement à l'intérieur des fonctions de `Dynarray`, la structure consiste en un enregistrement contenant une taille et un tableau dont la propre taille, appelée capacité, est supérieure ou égale à la première. Informellement :

```
forall (t : 'a dynarray). 0 <= t.length && t.length <= Array.length t.arr
```

2) Sûreté mémoire. L'implémentation de `Dynarray` est sûre. Autrement dit, *dans tous les cas*, usage correct ou non [6], le programmeur ne peut observer un accès mémoire erroné menant à la terminaison prématurée du programme. Ce deuxième critère est imposé par les garanties dynamiques du langage OCaml, d'ordinaire assurées par typage statique. Il n'est pas immédiat car notre implémentation exploite des fonctions non sûres (`Array.unsafe_get` et `Array.unsafe_set`), c'est-à-dire pour lesquelles il ne suffit pas d'être bien typé pour être sûr. Il faut en plus vérifier que les indices sont compris dans les bornes des tableaux.

Pour le vérifier, on raisonne dans un *cadre concurrent* en s'appuyant sur un *invariant faible* : à tout moment, la structure consiste en un enregistrement contenant une taille *positive*. Informellement :

```
forall (t : 'a dynarray). 0 <= t.length
```

On demande aussi que le tableau support `t.arr` respecte son propre invariant faible : c'est un tableau OCaml bien formé dont les éléments sont bien typés.

En sus de cet examen informel, nous avons mené un travail de vérification formelle dans la logique de séparation Iris (Jung, Krebbers, Jourdan, Bizjak, Birkedal, and Dreyer, 2018b), entièrement mécanisée dans le système Coq. Notre but était non seulement d'apporter des garanties supplémentaires mais aussi d'ajouter `Dynarray` à un corpus de structures

6. Nous considérons tous les contextes d'utilisation de `Dynarray` qui n'utilisent pas de fonctionnalités non-sûres. Mais on peut aussi étendre le raisonnement à tous les contextes qui sont sûrs contre toute implémentation entièrement sûre de `Dynarray`.

vérifiées utiles à de futurs développements. Nous décrivons ici le sous-ensemble traité et la méthode employée pour vérifier les deux critères. Les preuves mécanisées sont disponibles à https://github.com/clef-men/heap_lang/tree/jfla2024.

3.1 Fragment minimal

Le fragment minimal traité à ce jour se compose des fonctions suivantes :

```
val create : unit -> 'a dynarray
val make : int -> 'a -> 'a dynarray
val length : 'a dynarray -> int
val get : 'a dynarray -> int -> 'a
val set : 'a dynarray -> int -> 'a -> unit
val add_last : 'a dynarray -> 'a -> unit
val pop_last : 'a dynarray -> 'a
val fit_capacity : 'a dynarray -> unit
val reset : 'a dynarray -> unit
```

Ce fragment est suffisant pour parler d'usages concurrents problématiques, en particulier de conflits entre accès non sûrs au tableau support et rétrécissement de ce dernier par `fit_capacity` et `reset`.

Nous avons reproduit l'implémentation OCaml dans le langage HeapLang, l'instance canonique de la logique de séparation Iris. Sa syntaxe et sémantique sont relativement proches de celle d'OCaml. Nous l'utilisons car il est standard dans la communauté Iris et bien outillé.

Voici, par exemple, à gauche l'implémentation OCaml de `pop_last` et à droite l'implémentation HeapLang sans fard, telle qu'écrite en Coq.

```
let pop_last a =
  let {arr; length} = a in
  check_valid_length length arr;
  if length = 0 then raise Not_found;
  let last = length - 1 in
  match Array.unsafe_get arr last with
  | Empty ->
      Error.missing_element last length
  | Elem s ->
      Array.unsafe_set arr last Empty;
      a.length <- last;
      s.v
```

```
Definition dynarray_pop : val :=
  λ: "t",
    let: "len" := dynarray_length "t" in
    let: "arr" := dynarray_array "t" in
    assume ("len" <= array_length "arr") ;;
    assume (#0 < "len") ;;
    let: "last" := "len" - #1 in
    match: array_unsafe_get "arr" "last" with
    | None =>
        diverge #()
    | Some "ref" =>
        array_unsafe_set "arr" "last" &None ;;
        dynarray_set_size "t" "last" ;;
        !"ref"
    end.
```

Les deux implémentations diffèrent en plusieurs points (en plus des préférences de nommage différentes des deux auteurs, `a` ou `t` pour les tableaux dynamiques) :

1) Exceptions. Dans la version OCaml, on lève une exception sur un tableau vide et lorsqu'on détecte un usage erroné. HeapLang n'étant pas muni d'exceptions, le programme diverge en bouclant dans ces cas – dans une logique de *correction partielle* comme Iris, une boucle infinie est trivialement vérifiable. Néanmoins, de Vilhena and Pottier (2021) ont formalisé les effets algébriques et donc les exceptions en Iris. Nous espérons les ajouter un jour.

2) Représentation mémoire. La représentation des éléments du tableau support est moins efficace en HeapLang : le type `'a slot` est en quelque sorte remplacé par le type

`'a ref option`. S'accorder à la représentation OCaml ne pose toutefois pas de difficulté. Nous avons choisi la simplicité.

3) Modèle mémoire faible. Le modèle mémoire de HeapLang est fortement consistent alors que celui d'OCaml est faiblement consistent (Dolan, Sivaramakrishnan, and Madhavapeddy, 2018). Le modèle mémoire OCaml a aussi été formalisé en Iris dans le projet Cosmo (Mével, Jourdan, and Pottier, 2020). Il nous apparaît maintenant qu'il serait préférable d'utiliser Cosmo plutôt que HeapLang, et nous avons commencé à travailler dans cette direction.

4) Module `Array`. En lieu et place du module standard `Array`, la version HeapLang emploie des fonctions `array_*`. Ces fonctions et leurs spécifications ne sont pas axiomatisées. Nous les avons elles-même implémentées en HeapLang.

3.2 Correction fonctionnelle – correct tout seul

La vérification de la correction fonctionnelle de `Dynarray` en logique de séparation n'est pas novatrice. De façon classique, on définit une assertion logique `dynarray_model t vs` exprimant la possession unique de l'instance `t` contenant les valeurs `vs`. En Coq, cela ressemble à :

```
Definition dynarray_model t vs : iProp Σ :=
  ∃ l arr slots extra,
  ⌜t = #l⌝ ∗
  l.[length] ↦ #(length vs) ∗
  l.[array] ↦ arr ∗ array_model arr (slots ++ replicate extra &&None) ∗
  [∗ list] slot; v ∈ slots; vs, slot_model slot v.
```

`iProp Σ` est le type des assertions Iris. Cette définition décrit un enregistrement à deux champs. Le premier représente la taille de `vs`. Le second consiste en un tableau contenant les slots associés aux valeurs de `vs` suivis de valeurs `&&None` – ce qui permet d'arguer que la structure ne maintient pas ses anciens éléments en vie.

De façon classique toujours, on spécifie les fonctions de `Dynarray` à l'aide de triplets de Hoare. Typiquement, le prédicat `dynarray_model` y apparaît en précondition et postcondition en observant potentiellement un changement de valeurs contenues. Par exemple, les spécifications des fonctions `add_last` et `pop_last` s'écrivent :

```
Lemma dynarray_add_last_spec t vs v :
  {{{
    dynarray_model t vs
  }}}
    dynarray_add_last t v
  {{{
    RET #();
    dynarray_model t (vs ++ [v])
  }}}.
```

```
Lemma dynarray_pop_last_spec t vs v :
  {{{
    dynarray_model t (vs ++ [v])
  }}}
    dynarray_pop_last t
  {{{
    RET v;
    dynarray_model t vs
  }}}.
```

La spécification de `dynarray_add_last` se lit ainsi : étant donnée l'assertion `dynarray_model t vs` signifiant la possession unique de `t`, on peut procéder à l'appel et retrouver l'assertion où la liste de valeurs contenues a été augmentée de la valeur `v` donnée à la fonction. La spécification de `dynarray_pop_last` est symétrique.

3.3 Sûreté mémoire – sûr à plusieurs

En pratique, la vérification informelle de la sûreté mémoire n'apparaît pas difficile. Étant donné l'invariant faible déjà évoqué (taille positive, tableau support bien formé), il s'agit de

se convaincre que cet invariant potentiellement combiné à des tests dynamiques implique la validité de l'indice donné à `Array.unsafe_get` ou `Array.unsafe_set`.

Par exemple, dans l'implémentation de `pop_last` donnée plus haut, la bonne formation du tableau support, l'appel à `check_valid_length` et le test `if length = 0` (traduits par deux `assume` en HeapLang) suffisent à montrer que `last` est un indice valide pour `arr`. Le raisonnement reste local et simple, un humain peut aisément se convaincre de la sûreté.

Ceci étant dit, la question de la formalisation se pose. En particulier, intuitivement, l'invariant est plus faible mais parle de concurrence non synchronisée. Pour l'écrire, il nous faut donc une logique de séparation *concurrente* telle qu'Iris. La tâche se complexifie encore dans le cas d'un modèle mémoire faible.

En Iris, précisément, cette question a été posée dans les travaux sur RustBelt (Jung, Jourdan, Krebbers, and Dreyer, 2018a; Dang, Jourdan, Kaiser, and Dreyer, 2020; Matsushita, Denis, Jourdan, and Dreyer, 2022). Le problème dans sa forme générale est le suivant : dans un langage fortement et statiquement typé comme OCaml ou Rust, comment encapsuler un fragment – typiquement, une bibliothèque – non sûr.

En Rust, où cela se manifeste par des blocs `unsafe`, la réponse est : on encapsule du code non sûr derrière une interface sûre. C'est le cas, par exemple, de `std::vec`, qui expose la structure `Vec` comprenant des champs privés.

RustBelt formalise la chose à l'aide de la notion de *typage sémantique* : on interprète un type comme un prédicat en logique de séparation (relation logique). Pour justifier la sûreté d'un programme composé de parties sûres et de parties non sûres, on raisonne ainsi : 1) Sur le fragment sûr du langage, le typage syntaxique implique le typage sémantique. Les parties sans blocs `unsafe` acceptées par le typeur sont donc sémantiquement bien typées. 2) On peut composer des programmes sémantiquement bien typés pour obtenir un programme lui-même sémantiquement bien typé. 3) Le typage sémantique implique la sûreté (théorème d'adéquation d'Iris).

Autrement dit, pour vérifier la sûreté d'un programme utilisant des mécanismes non sûrs, on montre – manuellement ou de façon automatisée – que ce programme est sémantiquement bien typé. La combinaison avec tout autre programme sémantiquement bien typé, en particulier syntaxiquement bien typé, est alors sûre. C'est l'approche que nous avons adoptée.

Nous avons choisi pour OCaml une notion de type sémantique plus simple que celle de Rust. Nos types sémantiques sont définis en Coq par la typeclasse `iType` suivante :

```
Class iType (PROP : bi) (τ : val → PROP) := {
  #[global] itype_persistent v :: Persistent (τ v) ;
}.
```

`PROP` peut ici être pris comme l'ensemble des propositions Iris– c'est en quelque sorte une version plus bas niveau de `iProp Σ`. La partie importante est `itype_persistent`. Elle affirme la persistance du type sémantique τ au sens d'Iris, c'est-à-dire – pour faire court – le fait qu'une proposition soit toujours vraie, même quand l'état du programme change.

On définit par exemple le type sémantique `opt_type` τ, associé au type syntaxique `'a option`. Il s'agit d'exprimer qu'une valeur de ce type est soit de la forme `&&None`, soit de la forme `&&Some v` pour une valeur `v` de type τ.

```
Definition opt_type τ `{iType (iProp Σ) τ} t : iProp Σ :=
    ⌜t = &&None⌝ ∨ ∃ v, ⌜t = &&Some v⌝ ∗ τ v.
```

Le type `opt_type` τ ne traite pas tellement de ressources. Le type `reference_type` τ associé à `'a ref`, en revanche, affirme l'existence d'une cellule mémoire partagée dont la possession est stockée dans un *invariant* Iris.

```
Definition reference_type τ `{iType (iProp Σ) τ} t : iProp Σ :=
  ∃ (l : loc),
    ⌜t = #l⌝ ∗
    inv nroot (
```

```
    ∃ w,
    l ↦ w ∗ τ w
  ).
```

Le point remarquable dans cette définition est qu'on ne connaît pas la valeur contenue dans la cellule. On peut seulement accéder *atomiquement* à cette valeur et apprendre qu'elle est de type τ. On peut donc toujours lire et écrire une valeur de type τ dans la cellule.

Le type `dynarray_type` dirigeant notre vérification est à l'avenant :

```
Definition dynarray_type τ `{iType (iProp Σ) τ} t : iProp Σ :=
  ∃ l,
    ⌜t = #l⌝ ∗
    inv nroot (
      ∃ len cap arr,
        ⌜0 <= len⌝ ∗
        l.[length] ↦ #len ∗
        l.[array] ↦ arr ∗ array_type (slot_type τ) cap arr
    ).
```

C'est exactement la traduction formelle de l'invariant faible énoncé plus tôt : la structure consiste en un enregistrement contenant une taille *positive* et un tableau *bien formé* dont chaque élément constitue un objet valide pour le type `slot_type` τ.

Les spécifications *fortes* pour la correction fonctionnelle, `dynarray_model t vs`, sont alors remplacées par des spécifications *faibles*, `dynarray_type`τ`v` :

```
Lemma dynarray_push_type τ t v :
  {{{
    dynarray_type τ t ∗ τ v
  }}}
    dynarray_push t v
  {{{ u,
    RET u; unit_type u
  }}}.
```

```
Lemma dynarray_pop_type τ t :
  {{{
    dynarray_type τ t
  }}}
    dynarray_pop t
  {{{ v,
    RET v; τ v
  }}}.
```

Il ne s'agit de rien d'autre que de déclarations de type (sémantique) présentées sous la forme de triplets de Hoare : étant données une valeur `t` de type `dynarray_type` τ et une valeur `v` de type τ, la fonction `dynarray_push` renvoie une valeur de type `unit_type`.

La difficulté non apparente ici est que la preuve de ces spécifications se fait non pas dans un cadre essentiellement séquentiel où l'on possède exclusivement les ressources, mais dans un cadre concurrent où il faut continuellement interagir avec un invariant Iris. En particulier, l'implémentation de `dynarray_push` repose sur la fonction `array_blit` (notre implémentation de `Array.blit`) itérant sur une plage d'un tableau dont la spécification faible nous a conduit à généraliser sa spécifications forte de manière non triviale pour nous ramener à des accès atomiques au tableau.

Remarque 2. Il y a une différence de style entre les spécifications fortes et faibles. Une spécification forte `dynarray_model t vs` ne parle pas du type des éléments de `t` ; on pourrait le faire, mais ce n'est pas utile puisqu'on donne la valeur précise de tous les éléments, `vs`. Cette approche est plus expressive puisqu'elle permet de vérifier des programmes qui feraient des usage ≪mal typés≫ du tableau, mettant des éléments de type incompatible. (Par exemple si on voulait formaliser une implémentation utilisant `Obj.magic` pour les valeurs vides.) Au contraire, nos spécifications faibles parlent des types (sémantiques) des éléments. Ce n'est pas une préférence de notre part, c'est une nécessité pour réussir la formalisation. En effet, si on ne possède pas uniquement le tableau, on ne peut pas raisonner sur la valeur des éléments qui peut changer sous nos pieds. Il faut se mettre d'accord sur une spécification plus faible que leur valeur, que toutes les opérations préservent.

4 Discussion

4.1 Travaux futurs

Exceptions. Si le langage d'implémentation dans Iris avait des exceptions, nous pourrions représenter plus fidèlement le comportement OCaml. L'approche la plus simple serait d'avoir des spécifications qui ne parlent pas des exceptions, donc des triplets de Hoare dont la postcondition est vraie seulement si aucune exception n'est lancée. On pourrait aussi avoir une logique qui parle des exceptions dans les spécifications, par exemple en s'inspirant de de Vilhena and Pottier (2021).

Itérateurs. Nous n'avons formalisé que les problématiques liées aux usages concurrents de la structure de données. Nous comptons enrichir le sous-ensemble vérifié de `Dynarray` pour y ajouter notamment des fonctions d'itération telles que `iter`, `map` et `fold_left`. Cela nous amènera à la formalisation de l'invalidation des itérateurs – avoir des exceptions serait utile pour cela.

Permissions fractionnaires. Le modèle de programmation le plus courant en OCaml 5 est que les structures de données impératives peuvent être utilisées dans un cadre séquentiel, mais aussi dans un cadre concurrent "bien synchronisé", c'est-à-dire où les courses critiques sur la structure sont en lecture seule – les courses critiques où au moins un accès à une modification sont une erreur de programmation. Ce modèle de programmation s'applique à notre implémentation de Dynarray.

Par contre, les spécifications de correction fonctionnelle que nous avons formalisées demandent la possession unique du tableau dynamique, et ne permettent donc pas du tout les courses critiques, même en lecture seule. Il y a ici un écart entre les spécifications informelles et formelles. Pour capturer toute la spécification informelle, il faudrait utiliser des permissions fractionnaires dans la spécification formelle. Nous pensons que ce serait une extension facile de notre travail.

Cosmo. Une direction que nous avons déjà commencé à explorer consiste à adapter la vérification au modèle mémoire faible d'OCaml en utilisant Cosmo (Mével, Jourdan, and Pottier, 2020).

Nous nous attendons à des spécifications fortes essentiellement inchangées. En revanche, nous nous attendons à une petite difficulté quant aux spécifications faibles : les tableaux non atomiques nécessitent d'introduire une notion de *vue mémoire*. Intuitivement, il s'agit d'écrire que toutes les valeurs passées des éléments du tableau support sont du bon type sémantique. Nous pensons que cette propriété peut s'exprimer assez facilement dans la logique sous-jacente BaseCosmo.

Automatiser la sûreté. La correction fonctionnelle séquentielle demande souvent des spécifications fines et des arguments de preuve parfois sophistiqués, qui doivent être apportés par l'utilisateur. Par contre, la sûreté mémoire repose sur des invariants plus simples qui devraient pouvoir être automatisés dans leur écrasante majorité.

Il serait intéressant d'explorer des formes d'automatisation de cette partie des preuves. L'utilisateur fournirait des invariants faibles pour les structures de données sous forme de types raffinés, dans notre cas `0 <= t.length`. Cette approche pourrait tirer parti du langage de spécification Gospel.

4.2 Travaux liés

Autres implémentations. De nombreux langages proposent la structure de tableau dynamique dans leurs bibliothèques de base – c'est d'autant plus courant que le style du

langage est impératif. C'est par exemple la structure de base en Python, appelée `list`, mais Python a une implémentation purement séquentielle (comme OCaml 4) où la question de la sûreté concurrente ne se pose pas. C++ propose `std::vector`, qui n'est pas synchronisé – des usages concurrents cassent la sûreté mémoire. Le système de types de Rust interdit les usages non synchronisés de `std::vec`. Java propose deux versions, `ArrayList` (non synchronisé) et `Vector` (synchronisé) ; `ArrayList` n'utilise pas d'accès non-sûrs au tableau support, pour éviter les problèmes de sûreté mémoire.

Spécifications faibles en logique de séparation. Rendre formelle la notion d'≪invariant faible≫ utilisée pour écrire et relire le code de `Dynarray` nous a conduit naturellement à retrouver la notion de *type sémantique* du projet RustBelt (Jung, Jourdan, Krebbers, and Dreyer, 2018a; Dang, Jourdan, Kaiser, and Dreyer, 2020; Matsushita, Denis, Jourdan, and Dreyer, 2022). Ce n'est pas un hasard puisque la question est la même : vérifier du code `unsafe`.

Automatisation. Le projet RefinedRust propose de combiner vérification fonctionnelle et vérification de sûreté pour le langage Rust, avec automatisation. La présentation disponible en ligne discute justement du cas des tableaux dynamiques. Ce projet en cours aboutirait à un système proche de ce dont nous souhaiterions pour OCaml : par analogie, notre travail de vérification en Iris correspond au projet RustBelt, la couche d'automatisation que nous avons suggérée correspondrait à RefinedRust.

Remerciements Merci à François Pottier, et aux relecteurs et/ou relectrices anonymes des JFLA'24, pour leur relecture attentive et leurs remarques.

Références

Hoang-Hai Dang, Jacques-Henri Jourdan, Jan-Oliver Kaiser, and Derek Dreyer. Rustbelt meets relaxed memory. *Proc. ACM Program. Lang.*, 4(POPL), 2020.

Paulo Emílio de Vilhena and François Pottier. A separation logic for effect handlers. *Proc. ACM Program. Lang.*, 5(POPL), 2021.

Stephen Dolan, K. C. Sivaramakrishnan, and Anil Madhavapeddy. Bounding data races in space and time. In Jeffrey S. Foster and Dan Grossman, editors, *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*. ACM, 2018.

Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. Rustbelt: securing the foundations of the rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), 2018a.

Ralf Jung, Robbert Krebbers, Jacques-Henri Jourdan, Ales Bizjak, Lars Birkedal, and Derek Dreyer. Iris from the ground up: A modular foundation for higher-order concurrent separation logic. *J. Funct. Program.*, 28, 2018b.

Yusuke Matsushita, Xavier Denis, Jacques-Henri Jourdan, and Derek Dreyer. Rusthornbelt: a semantic foundation for functional verification of rust programs with unsafe code. In Ranjit Jhala and Isil Dillig, editors, *PLDI '22 : 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation, San Diego, CA, USA, June 13 - 17, 2022*. ACM, 2022.

Glen Mével, Jacques-Henri Jourdan, and François Pottier. Cosmo: a concurrent separation logic for multicore ocaml. *Proc. ACM Program. Lang.*, 4(ICFP), 2020.