

Étude d'une notion de cellule mutable stable et application à la vérification d'une structure de données due à Kaplan, Okasaki et Tarjan

Juliette Ponsonnet¹ encadrée par François Pottier²

¹ ENS de Lyon

² Inria Paris, équipe Cambium

Résumé La structure de deque proposée par Kaplan, Okasaki et Tarjan (2000) permet l'insertion et l'extraction par l'avant ou l'arrière ainsi que la concaténation. Elle utilise une forme d'état interne mutable tout en étant persistante, c'est-à-dire qu'elle paraît immutable de l'extérieur.

Nous proposons une implémentation en OCaml et HeapLang. Avec Iris, nous démontrons que cette implémentation est correcte même en contexte concurrent. Avec Iris et crédits-temps, nous démontrons que les opérations ont un coût amorti en $O(1)$ en contexte séquentiel.

The functional programmer sounds rather like a medieval monk, denying himself the pleasures of life in the hope that it will make him virtuous.

— John Hughes

Like knives, destructive updates can be dangerous when misused, but tremendously effective when used properly.

— Chris Okasaki

Plus ça change et plus c'est la même chose.

— Alphonse Karr

Table des matières

Étude d'une notion de cellule mutable stable et application à la vérification d'une structure de données due à Kaplan, Okasaki et Tarjan . <i>Juliette Ponsonnet encadrée par François Pottier</i>	1
1 Introduction.....	3
2 Préliminaires	4
2.1 Deques concaténables	4
2.2 Preuves de programmes	5
3 Implémentation OCaml.....	7
4 Références stables.....	9
4.1 Références concurrentes	9
4.2 Références séquentielles	10
5 Preuve formelle	12
5.1 Correction fonctionnelle	13
5.2 Complexité amortie.....	15
6 Travaux connexes	18
7 Conclusion	18
A Contexte du stage.....	21

1 INTRODUCTION

Une deque est une structure de données se comportant comme une file à deux bouts (*double-ended queue* en anglais), permettant d'enfiler et de défiler depuis l'avant et l'arrière. De telles structures sont utilisées dans des algorithmes de fenêtre glissante, et des variantes concaténables sont utilisées comme des représentations génériques d'une séquence de données (pour contenir des fragments de texte dans un programme d'édition par exemple). L'implémentation impérative canonique utilise une liste doublement chaînée et est très efficace. Cependant, l'aspect mutable de la structure peut être vu comme un défaut car l'édition de la structure invalide ses anciennes versions : elle n'est pas persistante.

La structure de deque proposée par Kaplan, Okasaki, and Tarjan [9] a de nombreux avantages car elle atteint de manière amortie la complexité optimale pour toutes les opérations qu'elle propose, et ce sans aucun compromis : elle est persistante [4] et peut être vue comme *purement fonctionnelle* dans un sens inhabituel du terme. Les mutations internes à la structure sont autorisées seulement si elles préservent les propriétés observables de la valeur. Par exemple, rééquilibrer un arbre binaire de recherche serait autorisé car ce changement n'est pas détectable par son interface abstraite. Ainsi, l'exécution effective de ces mutations ne peut pas changer le comportement d'un programme, mais seulement sa complexité. Ainsi on peut établir que cette structure est correcte dans un contexte concurrent sans synchronisation, mais on ne peut rien dire de la complexité de telles opérations.

Cependant, il n'existe aucune implémentation ni vérification formelle des algorithmes. La mutabilité employée par Kaplan, Okasaki, et Tarjan oblige l'utilisation d'outils sophistiqués comme Iris [8] car Rocq seul ne peut pas raisonner sur un état mutable. Nous proposons une implémentation [17] en OCaml et la vérification [15] formelle de ses propriétés les plus importantes. Dans ce rapport, nous discutons de l'implémentation et de ses différences avec la description par Kaplan, Okasaki, et Tarjan, puis introduisons la notion de référence stable qui modélise le type de mutabilité restreinte utilisée par les deques. Nous proposons une formalisation en Iris [8] des deques et analysons les opérations *push*, *pop*, *inject*, *eject* et *concat*. Pour chacune nous montrons qu'elle est correcte sous l'hypothèse de cohérence séquentielle (SC) : l'ordre relatif des opérations d'un même fil d'exécution est le même que celui du code source. Nous utilisons cette hypothèse car elle est présente dans Iris, mais nous pensons qu'elle n'est pas nécessaire. En principe, la même preuve pourrait être reproduite en Cosmo [3] pour montrer que même des cellules non-atomiques conviendraient. On notera que Kaplan, Okasaki, et Tarjan présentent leur algorithme dans un cadre séquentiel (une hypothèse beaucoup plus forte que SC), et que la correction concurrente est notre contribution. De plus, nous formalisons une analyse amortie de la complexité de ces cinq opérations en Iris avec crédit-temps [12] pour établir leur coût constant en interdisant certains accès. Précisément, si un appel à une des fonctions est en cours, aucune autre des fonctions ne peut être appelée. Enfin, nous montrons que le prédicat modélisant les deques est persis-



tant donc duplicable, soit que la structure, une fois créée, ne peut jamais être invalidée et que sa possession n’est pas exclusive.

La contribution de ce stage inclut donc l’implémentation OCaml de l’algorithme de Kaplan, Okasaki, et Tarjan, sa transcription manuelle en HeapLang, deux spécifications (une de correction concurrente et l’autre de correction et complexité séquentielle) en Iris, deux preuves vérifiées par la machine que l’implémentation vérifie les spécifications, et deux bibliothèques Iris décrivant une nouvelle notion de références stables, utilisées dans les preuves.

2 PRÉLIMINAIRES

2.1 Deques concaténables

Kaplan, Okasaki, and Tarjan [9] présentent une structure de données de *deque*, soit une structure permettant *push*, *inject*, *concat*, *pop* et *eject*, respectivement l’insertion au début et à la fin, la concaténation, et l’extraction au début et à la fin de la structure. Celle-ci est basée sur une arborescence de pointeurs

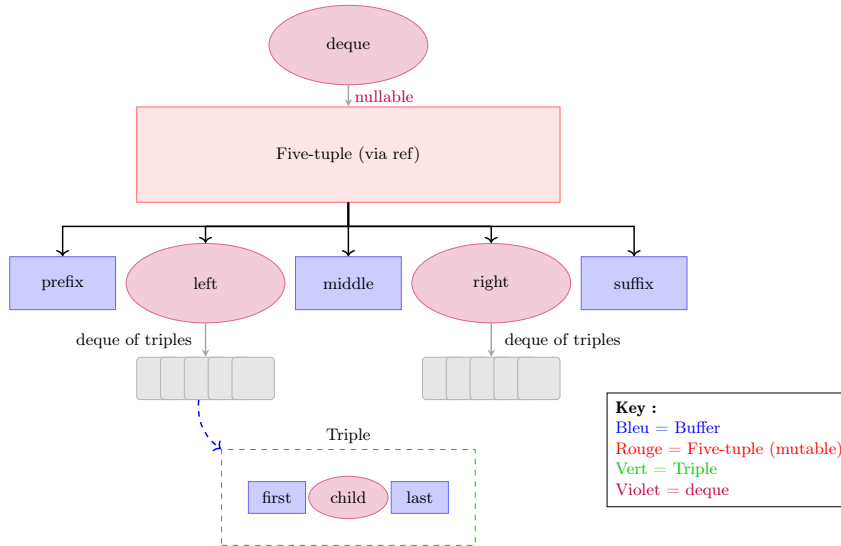


Figure 1. Représentation de la structure de Kaplan, Okasaki, et Tarjan

entre des structures intermédiaires (quintuplets ou *Five-Tuples* et triplets ou *triples*) contenant des tampons ou *Buffers*. Un tampon est une deque de taille comprise entre 0 et 8. Un tampon est persistant.

Leur représentation apparaît dans la Figure 1 et est comme suit : une deque d’éléments de X est un pointeur (potentiellement nul) vers un quintuplet. Un quintuplet contient un tampon préfixe, une deque fille-gauche, un tampon mi-

$$col\ n \triangleq \begin{cases} \text{red} & \text{si } n \in \{1, 8, 3, 6\} \\ \text{green} & \text{sinon} \end{cases}$$

$$pot\ p\ s \triangleq \begin{cases} pot\ s\ s & \text{si } p = 0 \\ 3 & \text{si } col\ p = col\ s = \text{red} \\ 1 & \text{si } \{col\ p, col\ s\} = \{\text{red}, \text{green}\} \\ 0 & \text{si } col\ p = col\ s = \text{green} \end{cases}$$

Figure 2. Potentiel d'un quintuplet en fonction de son préfixe et suffixe

lieu, une deque fille-droite et un tampon suffixe. Les tampons contiennent des éléments de X mais les deque contiennent des triplets. Un triplet contient un tampon premier, une deque fille-milieu et un tampon dernier. Similairement, les tampons contiennent des éléments de X mais la deque contient des triplets. Le niveau d'imbrication de triplets augmente avec la profondeur de la structure.

Une deque respecte plusieurs invariants, tous locaux. Il existe deux sortes de quintuplet : *suffixe seul* ou *équilibré*. Les tampons préfixe et milieu ainsi que les deque filles d'un quintuplet en suffixe seul sont vides, et le tampon suffixe est de taille comprise entre 1 et 8. Les tampons préfixe et suffixe d'un quintuplet équilibré sont de taille comprise entre 3 et 6, et son tampon milieu est de taille 2. Un triplet peut être *de gauche*, c'est-à-dire qu'il n'a pas d'enfant et que son tampon premier est de taille comprise entre 2 et 3, ou *équilibré*, c'est-à-dire que sa deque fille-milieu n'est pas vide et que ses deux tampons sont de taille comprise entre 2 et 3.

Les implémentations précises des diverses opérations sont décrites en prose par Kaplan, Okasaki, and Tarjan [9] et en OCaml dans la section §3. Dans leur présentation, on trouve une preuve que dans le cas séquentiel, toutes les opérations ont une complexité temporelle amortie en $O(1)$. Cette preuve utilise une analyse amortie par potentiel [19], que nous reprenons. Ainsi, avec le potentiel décrit en Figure 2, les appels récursifs sont compensés par la différence de potentiel global. En effet, il n'est pas possible de définir convenablement le potentiel d'une deque, car elles sont partagées comme filles d'autres deque. Kaplan, Okasaki, et Tarjan définissent donc le potentiel d'une famille de deque comme la somme des potentiels de tous les quintuplets en faisant partie. Nous montrons en section §5.2 qu'une analyse plus locale suffit, et qu'il n'y a donc pas besoin de définir le potentiel d'une famille de deque.

2.2 Preuves de programmes

Pour vérifier notre preuve, nous utilisons Iris [8]. Cet outil permet de vérifier des théorèmes sur les programmes. Pour cela, ces théorèmes doivent être exprimés dans une *logique de séparation* dont cette section introduit les éléments importants pour ce rapport. Des rappels sont donnés dans la section 2 de l'ar-

ticle [16] décrivant une autre structure mutable avec une interface persistante, les *thunks*. Nous ne détaillerons ici que les notions importantes à la compréhension de nos preuves.

Dans Iris, les assertions logiques sont affines : si elles sont utilisées comme hypothèses d’un théorème, elle sont consommées. De plus, elles ne peuvent pas être partagées. Par exemple, $\ell \mapsto v$, qui peut se lire comme “j’ai la connaissance exclusive que l’adresse ℓ contient v ”, ne peut pas exister en plusieurs exemplaires. On a alors une notion de compatibilité de propositions logiques. La conjonction séparante de P et Q , notée $P * Q$, est vérifiée lorsque P et Q sont vérifiées et compatibles. Si P' est une conséquence de P , alors $P' * Q$ est une conséquence de $P * Q$, c’est-à-dire que si P et Q sont compatibles, aucune modification légale de P ne peut contredire Q .

Certaines assertions n’ont pas autant de restrictions. Les assertions *persistentes* sont celles qui, une fois établies, sont vérifiées pour toujours. Elles ne peuvent pas être invalidées. Elles peuvent être partagées (entre différents fils d’exécution par exemple). Pour construire de telles assertions, Iris offre un mécanisme d’*invariants*. Le mot invariant admet plusieurs sens (invariant de structure [7], invariant de boucle [5]) mais celui-ci est encore différent. Un invariant peut être vu comme un contrat qui entre en vigueur à un certain instant de l’exécution du programme. Si P est une assertion, l’invariant associé à P peut être démontré en prouvant P , et à partir de cet instant la propriété P ne peut plus être invalidée, sauf d’une manière prévue par le contrat. L’utilisateur s’engage à respecter ce contrat. Comme les termes de ce contrat sont vrais pour toujours, l’invariant est persistant. Nous utilisons deux types d’invariants, les invariants *atomiques* et *non-atomiques*.

Un invariant atomique, noté \boxed{P} ³, s’assure que P est accessible partout et tout le temps. Sans restriction, on peut “ouvrir” l’invariant et en extraire P . Il faut alors, avant la prochaine étape de calcul, “refermer” l’invariant en fournissant une nouvelle preuve de P . Cela garantit que du point de vue de l’exécution du programme, la propriété P est toujours vérifiée, même si elle peut être momentanément brisée pendant le raisonnement de la preuve.

Un invariant non-atomique est plus permissif : l’assertion qu’il contient peut être invalide. Il existe un système de jetons. La garantie d’un invariant non-atomique est que si son jeton (les jetons sont nommés) est disponible, alors la propriété qu’il contient est vérifiée. Pour ouvrir un invariant non atomique il faut fournir son jeton, que l’on récupère en le refermant. Le nom non-atomique vient du fait que l’on peut effectuer des étapes de calcul entre l’ouverture et la fermeture de l’invariant. Cela permet des raisonnements plus complexes mais impose des restrictions plus fortes, car les jetons ne sont pas duplicables. Une preuve utilisant un invariant non-atomique ne pourra donc pas traiter le cas concurrent.

Pour les fonctions et structures récursives, Iris fournit la modalité “ \triangleright ” (se lit “*later*”), où $\triangleright P$ signifie que P sera vérifiée dans une étape de calcul. Une règle de raisonnement dite d’*induction de Löb* permet de raisonner sur des programmes

3. on omet ici les *namespaces* des invariants atomiques qui ne nous sont pas utiles

```

type 'a deque
val empty : 'a deque
val push  : 'a -> 'a deque -> 'a deque
val inject: 'a deque -> 'a -> 'a deque
val pop   : 'a deque -> 'a * 'a deque
val eject : 'a deque -> 'a deque * 'a
val concat: 'a deque -> 'a deque -> 'a deque

```


Figure 3. API OCaml des deques

récurifs : $(\triangleright P \multimap P) \vdash P$. De plus, si une définition mutuellement récursive de prédicats a la propriété que tout cycle contient la modalité “ \triangleright ”, alors elle peut être acceptée par Iris. Cela est utile car la définition de deque que nous utilisons en OCaml n’est pas acceptée par Rocq à cause de la condition de positivité.

Afin de modéliser le temps d’exécution, nous utilisons Iris avec crédits-temps (Iris[§]). Ce formalisme introduit une assertion \$1 qui représente un “crédit-temps”, c’est-à-dire la permission d’effectuer une unité de temps de calcul. Pour les utiliser, on ajoute dans le corps de chacune de nos fonctions une expression *tick* (), dont la spécification requiert la consommation de \$1 pour être appelée. Si toutes les fonctions qui ne sont pas marquées avec des *tick* s’exécutent en temps constant, le nombre de crédits-temps nécessaires à l’appel d’une fonction donne une majoration (à une constante près) de sa complexité. En effet, ces crédits étant affines, ils ne peuvent être dupliqués. De plus, ils ne peuvent pas être créés pendant l’exécution du programme. Ainsi ils apparaissent typiquement dans la précondition d’une fonction. Cependant, cela peut se faire indirectement dans les hypothèses pour permettre une analyse amortie.

Une partie des crédits-temps utilisés par la fonction peut se trouver dans l’assertion décrivant la structure de données. Notre définition de *deque* apparaissant dans la Figure 12 contient le potentiel de la structure sous forme de crédits-temps supplémentaires. On a alors une vraie analyse amortie [19] car une fonction créant une structure doit investir les crédits-temps nécessaires et une fonction utilisant la structure peut récupérer les crédits-temps qu’elle contient.

3 IMPLÉMENTATION OCAML

✎ L’article de Kaplan, Okasaki, and Tarjan [9] décrit deux types de deques. L’une des deux est plus simple mais n’est pas concaténable en temps amorti constant. Nous avons implémenté les deux [17], et les avons testées rigoureusement en utilisant Monolith [18]. Parmi les opérations implémentées, seules celles présentes dans la Figure 3. sont vérifiées. 

Le type abstrait α deque est paramétré par α le type de ses éléments. Ainsi, les deques filles stockent des triplets paramétrés par α , c’est un exemple typique de *nested data type* [1]. La constante *empty* est la seule représentation d’une deque vide satisfaisant les invariants. Le module définissant les deques est un foncteur OCaml. Il a comme argument un module définissant α *buffer*, un type

```

type 'a deque =
  'a nonempty_deque option
and 'a nonempty_deque =
  'a five_tuple ref
and 'a five_tuple = {
  prefix : 'a buffer;
  left   : 'a triple deque;
  middle : 'a buffer;
  right  : 'a triple deque;
  suffix : 'a buffer;
}
and 'a triple = {
  first  : 'a buffer;
  child  : 'a triple deque;
  last   : 'a buffer;
}

```

Figure 4. Définitions de types OCaml

de deque non-concaténables de taille au plus 8. La définition concrète du type deque de la Figure 4 calque la définition en prose de Kaplan, Okasaki, et Tarjan rappelée en section §2.1.

Le seul pointeur mutable est représenté par une référence OCaml (une cellule mémoire mutable), et chaque mise à jour remplace un quintuplet par un autre qui représente la même séquence d'éléments. Ainsi aucune fonction décrite en Figure 3 ne peut observer la mutabilité de la structure de données : elle est *purement fonctionnelle* du point de vue de l'utilisateur. Cependant, ces mises à jour sont nécessaires pour que la structure puisse à la fois être persistante et offrir une complexité amortie en $O(1)$.

Nous n'avons pas remarqué d'erreur dans l'article de Kaplan, Okasaki, et Tarjan, seulement une imprécision de langage par rapport à un branchement du programme. Un changement notable entre leur description et notre implémentation concerne les fonctions auxiliaires à *pop* et *eject*, où deux lectures consécutives du même pointeur (entre *inspect-first* et *pop-naïve*) sont fusionnées. Même si nous pensons que ce changement n'est pas nécessaire, cette double lecture n'a pas d'utilité et est probablement présente initialement pour alléger les descriptions des procédures. Contrairement à l'implémentation de Kaplan, Okasaki, et Tarjan, nous utilisons une unique référence pour le quintuplet au lieu de cinq, ce qui n'est pas important dans leur analyse mais nécessaire à la notre : il faut que les cinq champs soient modifiés simultanément, sans quoi l'algorithme n'est pas correct dans le cas concurrent.

Il est intéressant de remarquer que même si les invariants décrits en prose dans la section §2.1 sont toujours vérifiés par les valeurs visibles par l'utilisateur, il ne le sont pas toujours par les valeurs internes aux fonctions. Ainsi, certaines fonctions internes (comme *naïve-pop*) renvoient des deque illicites ou créent des deque même si leur argument est illicite (comme *push*). Nous modélisons cela

$$\begin{array}{lll}
\text{CSREF-PERSIST} & \text{CSREF-ALLOC} & \text{CSREF-READ} \\
\text{persistent}(\ell \Rightarrow \phi) & \{\phi v\} \text{ ref } v \ (\exists \ell) \ell \ \{\ell \Rightarrow \phi\} & \text{persistent}(\phi) \text{ -* } \\
& & \{\ell \Rightarrow \phi\} !\ell \ (\exists v) v \ \{\phi v\} \\
\\
\text{CSREF-WRITE} & & \\
\{\ell \Rightarrow \phi \text{ * } \phi v\} \ell := v \ () \ \{\} & &
\end{array}$$

Figure 5. Règles d'inférence des références stables concurrentes

par l'utilisation d'assertions plus faibles (comme $\ell \mapsto v$) pour les valeurs internes et des invariants Iris (avec les références stables) pour les arguments et valeurs renvoyées.

4 RÉFÉRENCES STABLES

*M*algré l'utilisation par Kaplan, Okasaki, et Tarjan de cellules mutables, les mises à jour ont toutes lieu dans un cadre non-destructif en un sens : l'ancienne valeur est équivalente à la nouvelle au sens de notre programme. Ainsi, si ℓ stocke la valeur v , on ne s'autorisera qu'à écrire dans ℓ une valeur v' qui sera équivalente à v , pour un sens d'équivalence bien choisi. Une implémentation possible serait donc de définir \equiv une relation d'équivalence et de n'autoriser l'écriture de v' en ℓ que si $v \equiv v'$. Cette méthode impose de se souvenir de la première valeur qu'a contenu la référence, alors qu'elle ne la contient probablement plus.

On peut choisir à la place de se souvenir de la classe d'équivalence de la valeur, sous la forme d'un prédicat $\lambda v'. v \equiv v'$. En utilisant la correspondance entre les prédicats sur les valeurs et les ensembles de valeurs, nous proposons de ne se souvenir que de la propriété importante que la valeur initiale vérifiait. Ainsi nous introduisons la notation $\ell \Rightarrow \phi$ (et des variantes) pour signifier que ℓ contient une valeur satisfaisant ϕ . On exigera également que $\ell \Rightarrow \phi$ soit persistante, ce qui obligera les mises à jour à respecter la propriété ϕ .

Les deux sous-sections suivantes présentent deux bibliothèques implémentant de telles références, on les dira *stables*. La première permet une utilisation sans contrainte, possiblement en contexte concurrent, mais limite les prédicats ϕ possibles. La seconde n'impose pas de restriction sur ϕ mais requiert une notion de possession pour les accès en écriture.

4.1 Références concurrentes

L'interface des références stables concurrentes apparaît sur la Figure 5. Elle décrit un prédicat $\ell \Rightarrow \phi$ qui représente une telle référence : il indique qu'à l'adresse ℓ se situe une valeur qui vérifie ϕ . Cette assertion est persistante au sens de Iris [8], contrairement à l'assertion usuelle $\ell \mapsto v$ qui doit être exclusivement possédée et peut être invalidée. On note que pour pouvoir utiliser pleinement

l'interface concurrente, le prédicat ϕ doit être persistant lui-même. Ce n'est pas un problème pour décrire une structure de données persistante, mais il peut être souhaitable de stocker de l'état fantôme (par exemple des crédits-temps qui représentent du potentiel) dans une référence stable, donc de privilégier le modèle de la section §4.2.

La règle CSREF-PERSIST exprime la persistance d'une référence stable. Une telle référence peut être créée et partagée librement par toute fonction ou tout fil d'exécution ayant accès à la mémoire. Ce partage est complètement sûr, ce qui montre que cette spécification est très forte. Pourtant, les autres règles laissent l'utilisateur libre d'utiliser les fonctions relatives aux références de manière similaire aux références classiques.

La règle CSREF-ALLOC décrit l'allocation d'une référence stable. Elle prend la forme d'un *triplet de Hoare* : une précondition, l'expression concernée, une quantification existentielle portant sur la valeur de retour et la postcondition. La même fonction que pour les références classiques sont utilisées, c'est une spécification alternative du même comportement. Elle énonce que si v vérifie ϕ , alors une référence peut être allouée comme stable pointant sur ϕ . On oublie v et "promet" de respecter la propriété. C'est cette promesse (ou invariant) qui est persistante.

La règle CSREF-READ décrit la lecture d'une référence stable. Comme précédemment, c'est une spécification alternative de la simple lecture d'une référence. En supposant que ϕ est persistante⁴ elle permet d'obtenir une valeur v vérifiant ϕ . La référence n'est pas invalidée par la lecture, mais n'est pas répétée dans la postcondition car persistante. L'hypothèse de persistance n'est pas contingente : si elle était omise, cette spécification permettrait de dupliquer et de faire persister la propriété ϕ quelle qu'elle soit, elle serait donc incohérente.

La règle CSREF-WRITE décrit l'écriture dans une référence stable. Sous l'hypothèse que v vérifie ϕ , une mise à jour de ℓ en v est permise, sans avoir à posséder une quelconque partie de la mémoire. La référence est toujours valide après l'écriture, mais n'est pas répétée dans la postcondition car persistante.

Nous proposons une définition concrète⁵ satisfaisant la spécification décrite ci-dessus à l'aide d'invariants atomiques.

Définition 1. $\ell \mapsto \phi \triangleq \boxed{\exists v. \phi \ v * \ell \mapsto v}$

Théorème 1. *La définition 1 vérifie toutes les règles d'inférence des références stables concurrentes.*

4.2 Références séquentielles

L'interface des références stables séquentielles apparaît en Figure 6 et décrit une interface alternative de références stables. Celle-ci n'impose aucune restric-

4. on note l'abus de notation, $\text{persistent}(\phi)$ est utilisé comme abréviation de $\forall v. \text{persistent}(\phi \ v)$

5. la définition en Iris ainsi que toutes les preuves se trouvent sur le dépôt kot-proof[15].



$$\begin{array}{lll}
\text{SSREF-NEW-POOL} & \text{SSREF-PERSIST} & \text{SSREF-ALLOC} \\
\Rightarrow \exists \pi. \mathfrak{z}^{\pi,0} & \text{persistent}(\ell \xRightarrow{\pi,n} \phi) & \{\phi v\} \text{ ref } v (\exists \ell) \ell \{\ell \xRightarrow{\pi,n} \phi\} \\
\\
\text{SSREF-READ} & & \\
(\forall v. \phi v \multimap \psi v) * \text{persistent}(\psi) \multimap & & \\
\{\ell \xRightarrow{\pi,n} \phi * \mathfrak{z}^{\pi,n}\} ! \ell (\exists v) v \{\psi v * \mathfrak{z}^{\pi,n}\} & & \\
\\
\text{SSREF-READ-WRITE} & & \\
\{\ell \xRightarrow{\pi,n} \phi * \mathfrak{z}^{\pi,n}\} ! \ell (\exists v) v \{\phi v * \mathfrak{z}^{\pi,n+1} * \forall v'. \text{once } \{\phi v' * \mathfrak{z}^{\pi,n+1}\} \ell := v' () \{\mathfrak{z}^{\pi,n}\}\} & &
\end{array}$$

Figure 6. Règles d'inférence des références stables séquentielles

tion sur ϕ mais requiert un système de possession. En effet, si la propriété stockée dans ℓ n'est pas persistante, on ne peut pas l'extraire et la remettre dans la référence. Ainsi l'interface demande à l'utilisateur de réinsérer une valeur qui satisfait ϕ . Avant cela, la référence est invalidée : c'est ce qu'expriment les *jetons* de possession. La structure de deque est naturellement stratifiée par sa profondeur, nous proposons donc une interface où les jetons sont indicés par des entiers naturels⁶, appelés niveaux. Un jeton $\mathfrak{z}^{\pi,n}$ est une assertion qui donne le droit d'accès à toutes les références de niveau supérieur à n . On peut interpréter π comme le nom de la région mémoire où se situent les références stables étiquetées par π .

La règle SSREF-NEW-POOL permet à tout instant d'allouer une nouvelle région π sur lequel on a tous les droits, grâce à $\mathfrak{z}^{\pi,0}$.

La règle SSREF-PERSIST est analogue à CSREF-PERSIST et exprime que même si une référence requiert un jeton pour être utilisée, elle peut être partagée et conservée.

La règle SSREF-ALLOC est analogue à CSREF-ALLOC. Elle permet d'allouer une référence stable dans n'importe quelle région et à n'importe quel niveau sans restriction.

La règle SSREF-READ-WRITE⁷ décrit le comportement d'une séquence lecture-écriture. L'utilisateur, en échange du jeton $\mathfrak{z}^{\pi,n}$, reçoit la valeur v à l'intérieur de la référence ainsi que la propriété ϕv et un jeton strictement plus faible $\mathfrak{z}^{\pi,n+1}$. De plus, il lui est possible de récupérer le jeton initial en rendant le jeton faible et en réassignant ℓ à une valeur v' vérifiant $\phi v'$.

La règle SSREF-READ émule CSREF-READ, car elle permet d'extraire de la référence $\ell \xRightarrow{\pi,n} \phi$ une conséquence persistante de l'assertion ϕ . Autrement dit, on peut récupérer v la valeur se situant en ℓ et une propriété ψ telle que $\phi v \vdash \psi v$ si ψ est persistante. Cependant, la lecture requiert le jeton $\mathfrak{z}^{\pi,n}$ car sans lui, rien ne garantit que la référence n'est pas actuellement invalidée par une séquence lecture-écriture.

6. il est possible d'utiliser des *namespaces* comme le font les invariants.

7. il est possible d'écrire une règle qui traite uniquement de l'écriture, mais elle n'est pas utile dans notre analyse

Nous proposons une définition concrète satisfaisant la spécification décrite ci-dessus à l’aide d’invariants non-atomiques. Afin de nommer ces invariants et gérer les jetons, nous définissons nos propres *namespaces*. La structure des

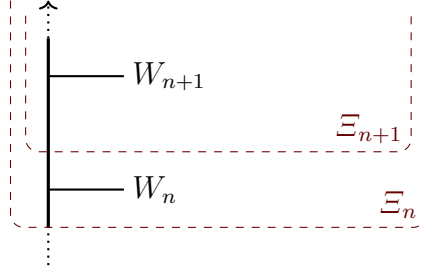


Figure 7. Hiérarchie des *namespaces*

namespaces apparaît en Figure 4.2 et prend la forme d’un peigne infini. C’est un détail technique permettant de stratifier les références (qui se situent aux niveaux W) et les jetons (étiquetés par Ξ).

Théorème 2. *Une définition analogue à la définition 1 mais utilisant les invariants non-atomiques vérifie toutes les règles d’inférence des références stables séquentielles.*

5 PREUVE FORMELLE

Deux spécifications incomparables sont vérifiées par l’algorithme de Kaplan, Okasaki, and Tarjan [9] : dans un contexte concurrent sans synchronisation, il est correct mais son coût en temps est logarithmique dans le pire des cas⁸, et dans un contexte avec synchronisation⁹, il a un coût constant amorti. Ainsi il n’existe pas d’unique triplet de Hoare en Iris qui décrive ces deux faits de manière satisfaisante, et nous proposons deux preuves différentes portant sur le même programme. Les preuves complètes en Iris se situent sur un dépôt annexe[15]. D’autres preuves, sur des versions simplifiées ou pures de l’algorithme, ont été réalisées pour appréhender les enjeux et anticiper les limitations possibles de Rocq & Iris.

Les preuves pures ont révélé une impossibilité à calquer le type OCaml décrit dans la Figure 4 car le type *deque* est en position négative, ce qui est un type de récursivité illicite en Rocq. En Iris, les points fixes gardés par la modalité “▷” fournissent une autre solution plus pratique à ce même problème. Cependant, nous avons été confrontés à des problèmes de performance de Rocq/Iris lors des

⁸. cette borne n’a pas été formellement vérifiée pendant le stage, mais des pistes ont été explorées

⁹. notre preuve est un peu plus générale que le cas séquentiel, voir la discussion sur les jetons dans la deuxième preuve

$$\begin{array}{l}
\text{Deque} : \text{Val} \rightarrow \text{list Val} \rightarrow \text{iProp} \qquad \text{DEQUE-PERSIST} \\
\qquad \qquad \qquad \text{persistent}(\text{Deque } d \text{ } xs) \\
\\
\text{DEQUE-EMPTY} \qquad \text{DEQUE-PUSH} \\
\text{Deque empty } [] \qquad \{ \text{Deque } d \text{ } xs \} \text{ push } x \text{ } d \text{ } (\exists d') \text{ } d' \{ \text{Deque } d' \text{ } ([x] ++ xs) \} \\
\\
\text{DEQUE-POP} \\
\{ \text{Deque } d \text{ } ([x] ++ xs) \} \text{ pop } d \text{ } (\exists d') \text{ } (x, d') \{ \text{Deque } d' \text{ } xs \} \\
\\
\text{DEQUE-CONCAT} \\
\{ \text{Deque } d \text{ } xs * \text{Deque } d' \text{ } xs' \} \\
\qquad \qquad \text{concat } d \text{ } d' \\
(\exists d'') \text{ } d'' \{ \text{Deque } d'' \text{ } (xs ++ xs') \}
\end{array}$$

Figure 8. Spécification de correction fonctionnelle

preuves des fonctions les plus complexes. Nous n'en avons pas identifié la cause exacte.

Une deque représente une séquence d'éléments de manière persistante. Nous la décrivons par un prédicat $\text{Deque} : \text{Val} \rightarrow \text{list Val} \rightarrow \text{iProp}$, où iProp est le type des assertions Iris, et Val est le type des valeurs Iris. L'assertion $\text{Deque } d \text{ } L$ signifiera que d est une deque bien formée **et** que cette deque représente la séquence de valeurs de L . Cette liste sert à formuler clairement les spécifications de correction des fonctions individuellement. Ce prédicat devra être persistant et compatible avec les fonctions *push*, *pop*, *inject*, *eject* et *concat*. Les listes seront notées avec une séquence d'éléments entre crochets et la concaténation de xs et ys sera notée $xs ++ ys$. On utilise aussi *flatten* qui à une liste de listes associe leur concaténation.

5.1 Correction fonctionnelle

La spécification que nous montrons pour la correction fonctionnelle apparaît dans la Figure 8 (mais ses arguments sont renversés pour des raisons d'application partielle). C'est une spécification purement fonctionnelle, où chaque fonction renvoie une nouvelle deque qui vérifie les propriétés souhaitées, sans invalider ses arguments car ils sont persistants.

La règle DEQUE-EMPTY indique que *empty* représente la liste vide, DEQUE-PERSIST que toutes les deques sont persistantes **et** que leur contenu ne change pas. On verra que cette spécification permet à la représentation interne d'une deque de changer, tant que ses invariants sont respectés et que la séquence représentée est constante. Les règles DEQUE-PUSH (et son symétrique pour *inject*), DEQUE-POP (et son symétrique pour *eject*) et DEQUE-CONCAT donnent les spécifications attendues en utilisant le prédicat.

Notre implémentation du prédicat *Deque* utilisé par la preuve de correction apparaît dans la Figure 10. Le prédicat *Deque* utilise les références stables concurrentes définies en section §4.1 pour modéliser les modifications restreintes.

$\frac{\text{TRIPLE-LEFT-LEANING} \quad c = 0 \quad x \in \{2, 3\} \quad y \in \{0, 2, 3\}}{\text{config}_3(x, c, y)}$	$\frac{\text{TRIPLE-HAS-CHILD} \quad c \neq 0 \quad x \in \{2, 3\} \quad y \in \{2, 3\}}{\text{config}_3(x, c, y)}$	$\frac{\text{5TUPLE-SUFFIX-ONLY} \quad s \in \{1 \dots 8\}}{\text{config}_5(0, 0, 0, 0, s)}$
$\frac{\text{5TUPLE-HAS-MIDDLE} \quad p \in \{3 \dots 6\} \quad s \in \{3 \dots 6\}}{\text{config}_5(p, l, 2, r, s)}$		

Figure 9. Prédicats de configuration locale

La propriété conservée est *fiveTuple L* qui oblige le quintuplet à rester bien formé et à représenter toujours la même liste.

Les prédicats *fiveTuple* et *triple* sont implémentés de manière similaire l'un à l'autre. Comme la représentation concrète des données n'est pas connue par le prédicat (une même séquence de données peut avoir plusieurs deque la représentant), nous utilisons une quantification existentielle sur les différentes composantes. Les lettres minuscules représentent des valeurs (tampons ou deque), les L indicés représentent les listes logiques correspondant aux valeurs, et les T indicés représentent les listes de triplets associés. Par exemple, une deque fille est une deque de triplets. Elle vérifie donc *deque T*, mais aussi chaque triplet $t \in T$ vérifie *triple l t* pour un certain l . La liste L associée à T sera donc la liste de ces l . Enfin, la liste représentée par la sous-structure est la concaténation des listes représentées par les tampons et des listes représentées par les triplets dans les deque filles.

Les invariants de structure sont exprimés sous forme de configurations locales, qui apparaissent dans la Figure 9. Une configuration est une contrainte simultanée sur la taille des listes représentées par les différentes composantes de la sous-structure. La règle TRIPLE-LEFT-LEANING décrit un triplet de gauche et la règle TRIPLE-HAS-CHILD décrit un triplet équilibré. Ces définitions sont identiques à celles données en prose dans la section §2.1. Leur utilisation est équivalente mais différente. Au lieu de vérifier si $|\text{flatten } L|$ est nul ou non, on utilise $|L|$ car aucun triplet n'est vide, et que toute deque fille contient des triplets. La règle 5TUPLE-SUFFIX-ONLY décrit un quintuplet en suffixe seul et la règle 5TUPLE-HAS-MIDDLE décrit un quintuplet équilibré. Comme pour les triplets, on ignore l'aplatissement des listes.

La présence de la modalité “▷” permet à cette définition mutuellement récursive d'être acceptée par Iris, même si cela requiert une utilisation technique de mécanismes¹⁰ internes à Iris.

La preuve de DEQUE-EMPTY est immédiate en définissant *empty* comme *NONE*. Similairement, DEQUE-PERSIST découle de la persistance des références stables. Les preuves de correction se font par induction de Löb (sauf pour *concat* qui n'est pas récursive). Comme tous les prédicats sont persistants, il s'agit de

10. on utilise des théorèmes de point fixe de morphismes contractifs

$$\begin{aligned} Deque\ L\ d &\triangleq \lceil d = NONE \wedge L = [] \rceil \vee \\ &\exists \ell. \lceil d = SOME(\ell) \rceil * \ell \Rightarrow fiveTuple\ L \end{aligned}$$

$$\begin{aligned} fiveTuple\ L\ d &\triangleq \exists p, l, m, r, s, L_p, L_l, L_m, L_r, L_s, T_l, T_r. \\ &\lceil d = (p, l, m, r, s) \wedge \\ &config_5(|L_p|, |L_l|, |L_m|, |L_r|, |L_s|) \wedge \\ &L = L_p ++ flatten\ L_l ++ L_m ++ flatten\ L_r ++ L_s \rceil * \\ &buffer\ L_p\ p * \\ &Deque\ T_l\ l * triples\ L_l\ T_l * \\ &buffer\ L_m\ m * \\ &Deque\ T_r\ r * triples\ L_r\ T_r * \\ &buffer\ L_s\ s \end{aligned}$$

$$\begin{aligned} triple\ L\ t &\triangleq \exists p, c, s, L_p, L_c, L_s, T_c. \\ &\lceil t = (p, c, s) \wedge config_3(|L_p|, |L_c|, |L_s|) \wedge \\ &L = L_p ++ flatten\ L_c ++ L_s \rceil * \\ &buffer\ L_p\ p * \\ &Deque\ T_c\ c * triples\ L_c\ T_c * \\ &buffer\ L_s\ s \end{aligned}$$

$$triples\ L\ T \triangleq \bigstar_i \triangleright triple\ L_i\ T_i$$

Figure 10. Prédicats décrivant les deques

dérouler tous les chemins d'exécution de la fonction en accumulant les informations sur les différentes variables intermédiaires. Une fois ces calculs faits, on rééquilibre les deques données en entrée, et il faut prouver avec CSREF-WRITE que cette écriture est légale. On alloue ensuite la valeur de retour avec la règle CSREF-ALLOC, qui demande de démontrer que le résultat vérifie bien la spécification.

5.2 Complexité amortie

La spécification de la Figure 8 doit être modifiée pour exprimer la complexité (ou le coût en crédits-temps) des opérations. Comme la preuve est mécanisée, il faut expliciter une majoration de cette quantité, ce que nous faisons sans démontrer son optimalité. De plus, pour exprimer le fait qu'on interdit les appels concurrents, on ajoute des jetons à la spécification. Cet ajout n'est pas ano-

$$\begin{array}{ll}
\text{DEQUE-PERSIST} & \text{DEQUE-EMPTY} \\
\text{persistent}(\text{Deque } \pi \ d \ xs) & \forall \pi. \ \text{Deque } \pi \ \text{empty} \ [] \\
\\
\text{DEQUE-PUSH} & \\
\left\{ \text{Deque } \pi \ d \ xs * \text{\textcolor{teal}{\$}}^\pi * \text{\textcolor{yellow}{\$}}7 \right\} & \\
\text{push } x \ d & \\
(\exists d') \ d' \ \left\{ \text{Deque } \pi \ d' \ ([x] ++ xs) * \text{\textcolor{teal}{\$}}^\pi \right\} & \\
\\
\text{DEQUE-POP} & \\
\left\{ \text{Deque } \pi \ d \ ([x] ++ xs) * \text{\textcolor{teal}{\$}}^\pi * \text{\textcolor{yellow}{\$}}168 \right\} & \\
\text{pop } d & \\
(\exists d') \ (x, d') \ \left\{ \text{Deque } \pi \ d' \ xs * \text{\textcolor{teal}{\$}}^\pi \right\} & \\
\\
\text{DEQUE-CONCAT} & \\
\left\{ \text{Deque } \pi \ d \ xs * \text{Deque } \pi \ d' \ xs' * \text{\textcolor{teal}{\$}}^\pi * \text{\textcolor{yellow}{\$}}56 \right\} & \\
\text{concat } d \ d' & \\
(\exists d'') \ d'' \ \left\{ \text{Deque } \pi \ d'' \ (xs ++ xs') * \text{\textcolor{teal}{\$}}^\pi \right\} &
\end{array}$$

Figure 11. Spécification de correction et de complexité

din : la nécessité de détenir un jeton pour pouvoir appeler une fonction impose l'impossibilité de deux appels simultanés. De plus, il n'existe par défaut aucun mécanisme pour communiquer un jeton d'un fil d'exécution à l'autre. Pour cela, il faudrait utiliser des mécanismes de synchronisation comme des verrous ou canaux. Le jeton $\text{\textcolor{teal}{\$}}^\pi$ sert de jeton générique à la famille de deque concernée et sera implémenté par $\text{\textcolor{teal}{\$}}^{\pi,0}$. La version modifiée apparaît dans la Figure 11, les coûts sont représentés sous forme de crédits-temps de Iris^{\$}, et ils sont fournis en quantité constante pour chaque appel. Cette spécification décrit bien une complexité constante.

L'analyse amortie est permise par notre ré-implémentation du prédicat *Deque*, qui pourra lui aussi contenir des crédits-temps. Un crédit temps n'est pas persistant, et il existe des moments dans l'exécution de l'algorithme où on a besoin de lire les crédits-temps dans la structure pour pouvoir créer la nouvelle structure (qui nécessite moins de crédits-temps). Pendant ce calcul intermédiaire, on ne dispose pas d'assez de crédits-temps pour rétablir l'invariant de potentiel, car l'ancienne valeur se situe encore dans la référence. Nous proposons donc d'utiliser des références stables séquentielles plutôt que concurrentes. Ce changement demande d'adapter les définitions des prédicats, les modifications apparaissent sur la Figure 12.

La référence stable concurrente devient séquentielle et ses arguments π et n sont passés à travers la structure. Nous choisissons que n représente la profondeur, ce qui permet d'utiliser le jeton $\text{\textcolor{teal}{\$}}^{\pi,n}$ pour lire une référence et ses filles simultanément, car elles se situent à des niveaux strictement supérieurs. Cette stratification est obligatoire pour étudier les appels récursifs de *push* et *pop* (et

$$\begin{aligned} deque \ \pi \ n \ L \ d &\triangleq \ulcorner d = NONE \wedge L = [] \urcorner \vee \\ &\exists \ell. \ulcorner d = SOME(\ell) \urcorner * \ell \xRightarrow{\pi \cdot \eta} fiveTuple \ n \ L \end{aligned}$$

$$\begin{aligned} fiveTuple \ \pi \ n \ L \ d &\triangleq \exists p, l, m, r, s, L_p, L_l, L_m, L_r, L_s, T_l, T_r. \\ &\ulcorner [\dots] \urcorner * \\ &\$ (pot \ k_p \ k_s) * \\ &buffer \ L_p \ p * \\ &deque \ \pi \ (n+1) \ T_l \ l * triples \ \pi \ n \ L_l \ T_l * \\ &buffer \ L_m \ m * \\ &deque \ \pi \ (n+1) \ T_r \ r * triples \ \pi \ n \ L_r \ T_r * \\ &buffer \ L_s \ s \end{aligned}$$

$$\begin{aligned} triple \ \pi \ n \ L \ t &\triangleq \exists p, c, s, L_p, L_c, L_s, T_c. \\ &\ulcorner [\dots] \urcorner * \\ &buffer \ L_p \ p * \\ &deque \ \pi \ n \ T_c \ c * triples \ \pi \ n \ L_c \ T_c * \\ &buffer \ L_s \ s \end{aligned}$$

$$triples \ \pi \ n \ L \ T \triangleq *_{i \triangleright} triple \ \pi \ n \ L_i \ T_i$$

Figure 12. Prédicats décrivant les deques avec le potentiel comme crédit-temps

leurs symétriques). On peut alors définir $Deque \ \pi \ d \ L \triangleq deque \ \pi \ 0 \ L \ d$ et démontrer la spécification.

L'enjeu principal de l'adaptation de la preuve précédente est la gestion des crédits-temps. Pour les récupérer, il faut utiliser SSREF-READ-WRITE afin de ne devoir rétablir l'invariant qu'à la prochaine écriture dans la référence. On remarquera que ceci n'est pas nécessaire ni possible pour *concat*, car elle lit deux références de même niveau, mais l'implémentation n'est pas récursive et ne demande pas d'utiliser le potentiel dans la référence. Il convient donc d'utiliser SSREF-READ pour extraire la structure et récupérer immédiatement le jeton. L'argument clef pour les fonctions récursives est de faire correspondre les branches d'exécution où il y a un appel récursif et celles où le potentiel de la structure diminue.

La plus grande subtilité se situe dans *pop* (et *eject*). En effet, dans les chemins d'exécution qui inspectent le premier élément d'une fille, il faut remarquer

que seuls ceux qui ne font pas d’appels récursifs font des appels coûteux à *push* et *concat*. Cette observation n’est pas immédiate car les deux instants sont relativement distants, et une preuve trop factorisée échouera à traiter les cas où des appels récursifs sont effectués. Au moment du branchement qui choisit entre l’appel de *naïve-pop* et *pop*, on ne peut pas céder la même quantité de crédits-temps aux deux branches ni utiliser le même prédicat sur la valeur de retour, car l’une est une deque valide et l’autre non. Il faut donc conserver les résultats dans une disjonction contenant le branchement choisi.

6 TRAVAUX CONNEXES

*S*ans utiliser de cellule mutable, Kaplan et Tarjan [10] proposent un autre algorithme de deque concaténable persistante de complexité constante non amortie. Celui-ci a été vérifié par Viennot et al. [20]. Malgré la ressemblance des structures de données, l’enjeu de cette preuve est la correction pure, et il n’y a pas besoin d’outils comme Iris. D’autres articles présentent des vérifications d’algorithmes avec Iris[§], comme *Union-Find* [2] ou les *transient stacks* [11]. Cette approche avec de la logique de séparation peut être amenée à un cadre industriel comme l’ont fait Haslbeck et Lammich [6] en vérifiant la complexité d’un algorithme de tri aussi performant que celui de la bibliothèque standard de C++.

D’autres travaux ont des ressemblances plus spécifiques. Les *thunks* [16] sont une autre structure de donnée mutable dont la spécification est persistante. Comme les deques, lors de leur lecture, il peut y avoir un calcul supplémentaire (évaluation de la valeur) selon l’état interne de la structure, et ce de manière invisible pour l’utilisateur.

Des approches différentes de la logique de séparation sont aussi possibles pour l’analyse de complexité amortie, comme le *framework* proposé par Nipkow [13], qui simplifie l’étude des structures purement fonctionnelles.

7 CONCLUSION

*N*otre preuve mécanisée de l’algorithme de Kaplan, Okasaki, et Tarjan confirme leurs théorèmes et leurs méthodes : ils affirment utiliser une “méthode standard basée sur les crédits”, en opposition à la méthode des débits créée par Okasaki [14] pour étudier des algorithmes utilisant des *thunks*. Nous utilisons précisément ces outils, les crédits étant représentés par des assertions affines Iris[§]. Contrairement à leur analyse, nos arguments de potentiel sont complètement locaux : nous n’avons jamais à définir le potentiel d’une famille de deques.

Notre preuve de correction a des hypothèses très relâchées. Même si la structure était initialement présentée dans un contexte séquentiel, nous démontrons le cas concurrent et nous pensons que notre seule hypothèse (la cohérence séquentielle) reste trop forte. En effet, pour la correction, la valeur lue dans la référence n’a pas d’importance, car toutes les anciennes valeurs écrites dans la référence vérifient la même propriété.

Notre preuve de complexité échoue dans un contexte concurrent. Nous pensons que la complexité dans le pire des cas est logarithmique, et une preuve formelle de ce fait mettrait en lumière les cas qui imposent du travail supplémentaire à l'algorithme. Nous utilisons des jetons comme on pourrait utiliser un verrou afin d'éviter une course critique. Il serait intéressant d'étudier les différentes possibilités pour les jetons, et d'introduire la possibilité de les transférer entre les fils d'exécution, ou de les attendre avec un verrou.

L'utilisation de mutations restreintes par Kaplan, Okasaki, and Tarjan [9] nous a mené à écrire deux bibliothèques de références stables, qui semblent être un concept applicable à l'analyse d'autres algorithmes comme les arbres évasés (*splay trees*) ou en général les *self-adjusting data structures*.

L'objectif du stage est rempli¹¹, car la structure de données a pu être implémentée entièrement et efficacement, nous avons pu reproduire et vérifier l'analyse de Kaplan, Okasaki, et Tarjan, généraliser leur raisonnement pour étendre leurs théorèmes, tout en isolant le concept de "cellule mutable stable" que nous formalisons dans deux bibliothèques de références stables.

Remerciements Merci à François Pottier de m'avoir encadrée et aidée dans ce projet, à Hugo Segoufin-Chollet d'avoir apporté des cookies aux goûters, à toute l'équipe Cambium avec qui c'était un plaisir de partager des repas et des discussions, et en particulier à Yannick Zakowski qui m'a recommandé ce stage.

Références

1. Bird, R., Meertens, L. : Nested datatypes. In : Mathematics of Program Construction (MPC). Lecture Notes in Computer Science, vol. 1422, pp. 52–67. Springer (1998), <http://www.cs.ox.ac.uk/richard.bird/online/BirdMeertens98Nested.pdf>
2. Charguéraud, A., Pottier, F. : 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 (Mar 2019), <http://cambium.inria.fr/~fpottier/publis/chargueraud-pottier-uf-slrc.pdf>
3. Di Cosmo, R. : Deciding type isomorphisms in a type assignment framework. *Journal of Functional Programming* **3**(3), 485–525 (1993), <http://www.dicosmo.org/Articles/JFP94.dvi>
4. Driscoll, J.R., Sarnak, N., Sleator, D.D., Tarjan, R.E. : Making data structures persistent. *Journal of Computer and System Sciences* **38**(1), 86–124 (1989), [https://doi.org/10.1016/0022-0000\(89\)90034-2](https://doi.org/10.1016/0022-0000(89)90034-2)
5. Floyd, R.W. : Assigning meanings to programs. In : Mathematical Aspects of Computer Science. Proceedings of Symposia in Applied Mathematics, vol. 19, pp. 19–32. American Mathematical Society (1967), <https://people.eecs.berkeley.edu/~necula/Papers/FloydMeaning.pdf>

11. même si une partie de la programmation a été raffinée ou terminée après la date de fin du stage, dont la preuve complète de *pop* à cause des problèmes de performance

6. Haslbeck, M.P.L., Lammich, P. : For a few dollars more - verified fine-grained algorithm analysis down to LLVM. In : European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 12648, pp. 292–319. Springer (Mar 2021), https://www21.in.tum.de/~haslbema/documents/Haslbeck_Lammich_LLVM_with_Time.pdf
7. Hoare, C.A.R. : Proof of correctness of data representations. *Acta Informatica* **4**, 271–281 (1972), <http://dx.doi.org/10.1007/BF00289507>
8. Jung, R., Krebbers, R., Jourdan, J.H., Bizjak, A., Birkedal, L., Dreyer, D. : Iris from the ground up : A modular foundation for higher-order concurrent separation logic. *Journal of Functional Programming* **28**, e20 (2018), <https://people.mpi-sws.org/~dreyer/papers/iris-ground-up/paper.pdf>
9. Kaplan, H., Okasaki, C., Tarjan, R.E. : Simple confluent persistent catenable lists. *SIAM Journal on Computing* **30**(3), 965–977 (2000), <http://www.aladdin.cs.cmu.edu/papers/pdfs/y2000/catenable.pdf>
10. Kaplan, H., Tarjan, R.E. : Purely functional, real-time deques with catenation. *Journal of the ACM* **46**(5), 577–603 (1999), <http://www.math.tau.ac.il/~haimk/adv-ds-2000/jacm-final.pdf>
11. Moine, A., Charguéraud, A., Pottier, F. : A high-level separation logic for heap space under garbage collection. *Proceedings of the ACM on Programming Languages* **7**(POPL), 718–747 (Jan 2023), <https://doi.org/10.1145/3571218>
12. Mével, G., Jourdan, J.H., Pottier, F. : Time credits and time receipts in Iris. In : European Symposium on Programming (ESOP). Lecture Notes in Computer Science, vol. 11423, pp. 1–27. Springer (Apr 2019), <http://cambium.inria.fr/~fpottier/publis/mevel-jourdan-pottier-time-in-iris-2019.pdf>
13. Nipkow, T., Brinkop, H. : Amortized complexity verified. *Journal of Automated Reasoning* **62**(3), 367–391 (2019), <https://www21.in.tum.de/~nipkow/pubs/jar18.pdf>
14. Okasaki, C. : *Purely Functional Data Structures*. Cambridge University Press (1999), <https://doi.org/10.1017/CB09780511530104>
15. Ponsonnet, J., Pottier, F. : kot-proof. Online repository (Jul 2025), <https://github.com/HiiGHoVuTi/kot-proof>
16. Pottier, F., Guéneau, A., Jourdan, J.H., Mével, G. : Thunks and debits in separation logic with time credits. *Proc. ACM Program. Lang.* **8**(POPL) (Jan 2024). <https://doi.org/10.1145/3632892>, <https://doi.org/10.1145/3632892>
17. Pottier, F., Ponsonnet, J. : kot. Online repository (Jul 2025), <https://github.com/fpottier/kot>
18. Pottier, F. : Strong automated testing of OCaml libraries. In : *Journées Françaises des Langages Applicatifs (JFLA)* (Feb 2021), <http://cambium.inria.fr/~fpottier/publis/pottier-monolith-2021.pdf>
19. Tarjan, R.E. : Amortized computational complexity. *SIAM Journal on Algebraic and Discrete Methods* **6**(2), 306–318 (1985), <http://dx.doi.org/10.1137/0606031>
20. Viennot, J., Wendling, A., Guéneau, A., Pottier, F. : Verified purely functional catenable real-time deques (2025), <https://arxiv.org/abs/2505.07681>

A CONTEXTE DU STAGE

J'ai fait mon stage à l'INRIA Paris où travaillent 700 personnes réparties en 32 équipes, dont l'équipe Cambium qui m'a accueillie. J'étais encadrée par François Pottier, qui était mon contact privilégié lorsque j'avais des questions et avec qui j'ai réfléchi pour avancer sur les problématiques du stage. Dans mon bureau il y avait deux autres stagiaires de L3 travaillant avec Rocq, et nous échangeons beaucoup sur nos sujets, nos problèmes et nos idées. J'ai pu échanger avec le reste de l'équipe quotidiennement de manière informelle pendant les pauses et en particulier sur le temps de midi. De plus, des réunions pseudo-hebdomadaires avaient lieu au sein de l'équipe pour décrire les sujets de travail de chacun et être au courant de ce que font les autres. C'était pratique pour avoir une idée de ce que font les chercheurs dans ce domaine, mais aussi pour savoir avec qui discuter si on a des idées ou des questions.