Stage L3: Formalisation des dictionnaires en Coq

Valeran MAYTIE

Juillet 2023

1 Structure d'accueil

Mon stage se déroule au LMF (Laboratoire de Méthodes Formelles) [1] dans l'équipe Toccata [2]. C'est une équipe de recherche du centre Inria Saclay-Île-de-France. Celle-ci est composée de 7 membres permanents. Parmi eux se trouve Guillaume Melquiond, mon encadrant de stage. Ses travaux de recherche se situent à l'intersection des domaines de l'arithmétique des ordinateurs et de la preuve formelle.

2 Contexte scientifique

2.1 Présentation Général

Coq est un assistant de preuve basé sur de la théorie de types. De ce fait, il possède un langage de programmation qui peut être extrait vers OCaml ou directement interprété. Le projet ERC Fresco [3] vise à transformer ce langage en un outil rapide de calcul formel. Un élément clé est la conception d'un langage de programmation dédié ainsi que des structures de données de haut niveau. Le but de ce stage est de se pencher sur les structures de données associatives en utilisant les tableaux persistants ajoutés en 2010 à Coq.

Comme Coq possède un langage il est intéressant de chercher des moyens d'améliorer les performances des structures de données tout en préservant la validité des preuves. Cela permettrait d'élargir les possibilités d'utilisation de Coq dans des projets où la performance est critique.

2.2 Les enjeux

En Coq, la performance a souvent été sacrifiée, les programmes sont souvent extraits vers OCaml donc il y a peu d'intérêt à créer des structures performantes directement dans ce langage. Toutefois, de nos jours, Coq est largement répandu et est utilisé dans des projets qui exploitent directement son interpréteur. Aujourd'hui les seules structures de dictionnaires disponibles sont les FMapAVL qui sont difficiles à prendre en main et peu performantes. Une implémentation de table de hachage est disponible dans la bibliothèque standard de Coq, elle a été faite à partir d'arbre de Patricia [4]. Cependant une structure de données plus rapide serait bénéfique pour des opérations telles que la mémoïsation ou le partage maximal [BJM14].

Il est également très intéressant d'étudier si l'utilisation de structures impératives persistantes peut améliorer l'efficacité des programmes. En effet, il n'a pas été mentionné que l'utilisation de structures persistantes augmentera automatiquement l'efficacité. Il est possible que ces tableaux utilisent beaucoup de mémoire, ce qui peut entraîner des temps d'allocation plus longs. De plus, le temps d'exécution du *garbage collector* (ramasse-miettes) peut également augmenter.

1. https://lmf.cnrs.fr/
2. https://toccata.gitlabpages.inria.fr/toccata/index.fr.html
3. https://fresco.gitlabpages.inria.fr/
4. https://coq.inria.fr/library/Coq.FSets.FMapPositive.html

2.3 Travail effectué

Le but du stage est d'explorer différentes implémentations de dictionnaires basées sur des tableaux en Coq. Le premier travail réalisé a consisté à comprendre comment les tableaux sont implémentés en Coq en se référant à la littérature existante [AGST10]. Les entiers machines sont utilisés par les tableaux pour des raisons d'efficacité : il est donc important de bien comprendre leur fonctionnement. L'implémentation des tableaux utilise la structure persistante de Baker. Pour approfondir mes connaissances, j'ai lu l'article de Baker lui-même [Bak91], ainsi que celui de Sylvain Conchon et Jean-Christophe Filliâtre [CF07], afin de compléter le cours donné par Xavier Leroy au Collège de France.

2.3.1 Tables de Hachage

Avant de me lancer dans l'implémentation des tables de hachage, j'ai utilisé des dictionnaires basés sur des arbres de Patricia afin d'avoir une idée des spécifications des fonctions de base. Ensuite, en me renseignant sur les différentes implémentations des tables de hachage, j'ai pensé qu'il serait plus simple de commencer par implémenter les tableaux avec une résolution des collisions à l'aide de listes chaînées.

J'ai utilisé les tableaux de Coq pour faire une première implémentation naïve de table de hachage pour avoir une première base de travail. Pour la résolution de collision j'ai utilisé des seaux, c'est-à-dire que les cases du tableau sont composées de liste chainée, donc si deux éléments avec deux clés différentes sont attribués à la même case alors ils sont tous les deux ajoutés à la liste.

Ainsi, j'ai défini le type des tables de la manière suivante :

```
Inductive bucket : Set :=
  | Empty : bucket
  | Cons (hash: int) (key: A) (value: B) (next: bucket) : bucket.

Record t : Set := hash_tab {
  size : int;
  hashtab :   PArray.array bucket;
}.
```

Ensuite, j'ai défini les fonctions de base : `add`, `empty`, `find`. J'ai choisi de faire des tables de hachage similaires à celles d'OCaml, c'est-à-dire de cacher les anciennes valeurs d'une clé à chaque ajout. À cause de ce choix j'ai défini une nouvelle fonction `findall` qui donne toutes les valeurs associées à la même clé. J'ai donc fait mes premières preuves sur les tableaux en spécifiant ces fonctions.

```
Lemma add_same: forall k (h: t B) v,
  find_all (add h k v) k = v :: (find_all h k).

Lemma add_diff: forall k k' (h: t B) v,
  k' <> k → find_all (add h k v) k' = find_all h k'.
```

La plupart des fonctions récursives sont écrites en récursive terminale. Pour pouvoir faire des preuves par récurrence j'ai écrit les mêmes fonctions sans faire d'optimisation. Ensuite, j'ai prouvé qu'elles ont le même résultat pour pouvoir faire des réécritures dans les preuves et enfin faire une récurrence.

2.3.2 Tableaux dynamique

Dans un langage impératif pour avoir une table de hachage efficace il faut utiliser des tableaux dynamiques. Je vais donc tester si faire cette transformation va rendre la structure plus efficace.

J'ai décomposé le problème en 3 fonctions :

— `rehash_bucket`: récupère les éléments d'une case pour les mettre dans la nouvelle table.
— `copy_tab`: copie un tableau dans un nouveau tableau.
— `resize`: va redimensionner la table de hachage

J'ai spécifié `resize` avec cette formule :

```
Lemma find_all_resize:
  forall (h: t) (k: A),
  find_all (resize h) k = find_all h k.
```

Cette spécification a été très difficile à prouver, car `resize` utilise deux autres fonctions externes récursives. Pour commencer, j'ai attribué certaines propriétés aux deux autres fonctions, qui ont été admises initialement pour m'assurer de leur utilité par la suite. Une fois que cette étape de preuve a été réussie, j'ai prouvé toutes les assertions liées à la fonction en question, parfois sans disposer de suffisamment d'hypothèses. Pour mener à bien ces preuves, j'ai également ajouté des conditions dans le programme afin de vérifier si un élément se trouve effectivement dans le bon seau.

Dans la preuve de correction des fonctions j'ai rencontré une difficulté : un utilisateur pourrait fournir n'importe quelle table en paramètre des fonctions, et je n'ai donc aucune information sur la valeur par défaut de la table ou si les éléments sont dans le bon saut (complication pour `findall`). Deux solutions étaient possibles : soit établir un invariant sur la structure de données, soit ajouter des conditions dans le programme pour gérer les cas impossibles dans les preuves. J'ai choisi la deuxième solution, car la création d'un invariant aurait impliqué la création de termes de preuve très complexes, ce qui aurait entraîné une perte de la rapidité des entiers machine. Souvent, il aurait fallu passer par des représentations de nombres peu performantes dans les preuves. De plus, il aurait fallu avoir un historique des écritures dans la table ce qui aurait empêché le *garbage collector* de libérer la mémoire.

Par exemple, pour la fonction `rehash_bucket` j'ai rajouté ce test en plus :

```
Definition rehash_bucket (new_tab last_tab: table) (new_size last_size i: int) : table :=
  fold_right (fun h k v a ⇒
    if i =? key_index last_size h then  (* test si l'element est dans le bon seau *)
      let h_b := key_index new_size h in
      a.[ h_b ← Cons h k v a.[h_b]] (* copie dans le nouveau tableau *)
    else a)
  new_tab last_tab.[i].
```

2.3.3 Tests réalisés

Au départ, j'ai cherché à écrire des fonctions qui pourraient bénéficier d'une optimisation par mémoïsation. J'ai immédiatement pensé à la fonction de Fibonacci. Cependant, en effectuant cette optimisation, la complexité devient linéaire et le résultat augmente de manière exponentielle, ce qui entraîne rapidement des débordements. J'aurais pu lancer plusieurs fois la fonction mais au lieu de ça j'ai cherché une fonction memoïsante plus difficile à calculer. J'ai écrit une fonction qui calcule les coefficients binomiaux à l'aide du triangle de Pascal. Ma fonction va faire $\mathcal{O}(n^2)$ accès à la table. Pour la memoïsation, j'ai comme type de clé un couple d'entier et comme valeur un entier. Il faut aussi créer une fonction de hachage qui éviterait le plus possible les collisions : j'ai choisi d'utiliser $h(k1, k2) = k1 + k2 \times n$ (n une constante). Il a fallu aussi faire une version avec une paire de *positive*[5] car les arbres de Patricia vont utiliser ce type. J'ai utilisé la même fonction puis transformé l'entier machine en *positive*, cette transformation est de complexité $\mathcal{O}(\log(n))$ donc un peu coûteuse. Il y a donc deux versions du test : une avec des clés qui ont une fonction de hachage couteuse et l'autre avec une fonction de hachage gratuite. J'ai aussi fait le test avec les *FMap* de la bibliothèque standard de Coq[6].

J'ai pu mesurer de très bonnes performances pour les tables de hachage sur les entiers Tableau-1. "Table int" et "Patricia" représentent les deux structures créées avec une fonction de hachage gratuite, contrairement à "Table Nat" et "Patricia Nat" qui utilisent une fonction de hachage coûteuse.

5. Représentation d'un nombre en binaire avec une liste chainée
6. https://coq.inria.fr/library/Coq.FSets.FMapAVL.html

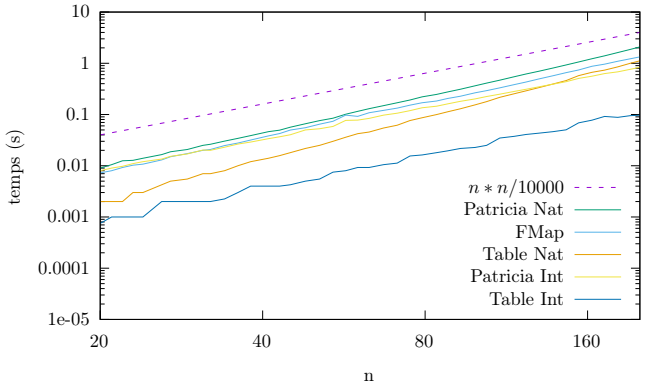

FIGURE 1 – Temps d'exécution de pascal_memo($2n$, n) dans un repère log-log

J'ai aussi fait une fonction qui vérifie la conjecture de Syracuse jusqu'à un certain entier n. J'utilise des dictionnaires pour retenir les valeurs déjà croisées. On leur associe l'entier de départ de la suite. Grâce à cet enregistrement je peux vérifier si j'ai trouvé une boucle (arrêt du calcul) ou si la valeur a déjà été rencontrée (passage à la prochaine valeur). Les tests de performances ont été effectués avec deux types de valeur numérique : les entiers machines et les *positive*. J'ai testé avec mon implémentation des arbres de Patricia, nos tables de hachage et les *FMap* de Coq. J'ai aussi utilisé les *PositiveMap*[7] ("Fmap Pos") de Coq qui sont aussi des arbres de Patricia.

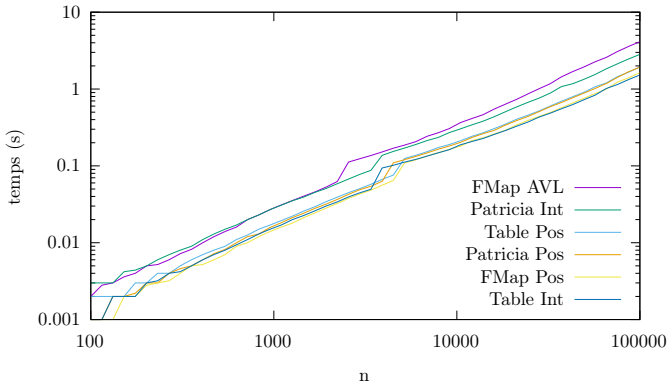

FIGURE 2 – Temps d'exécution de Syracuse_test(n) dans un repère log-log

De ces deux graphes on peut conclure que les tables de hachage sont largement plus performantes que des arbres pour des calculs mémoïsant faisant intervenir uniquement des entiers machines. Sinon si les fonctions utilisent d'autres types de donnée comme les *positive* le coût de conversion vers les entiers machines se ressent dans le temps d'exécution.

Pour la suite du stage il serait utile de faire de nouveaux tests en essayant de pousser les optimisations des fonctions avec les arbres. Par exemple, passer avec les *positive* pour le calcul des coefficients binomiaux. Ensuite, il faudra que je fasse une librairie Coq pour rendre les tables de hachage utilisables pour tous.

7. https://coq.inria.fr/library/Coq.FSets.FMapPositive.html

3 Retour d'expérience

Ce stage a été une excellente opportunité pour apprendre de nouvelles choses et approfondir mes connaissances dans certains domaines. J'ai pu réaliser ce stage grâce au TER que j'ai effectué d'octobre à avril, et qui a été l'un des "cours" les plus utiles pour mener à bien ce stage. En travaillant sur CompCert, j'ai appris à utiliser Coq, et pour apprendre à réaliser des preuves, j'ai pu consulter le livre "Coq'Art" de Bertot et Casteran[BC15], qui m'a généreusement été prêté par Christine Paulin-Mohring durant le TER et le début du stage. Le cours de génie logiciel avancé m'a également beaucoup aidé, car il m'a permis de comprendre l'importance des spécifications et de connaître les outils disponibles pour vérifier ces propriétés parfois très complexes. Le cours de lambda calcul a également été très utile pour mieux comprendre ce qui se passe dans Coq. Il m'a donné des bases solides pour être plus à l'aise lors de la lecture d'articles, car le lambda calcul est un concept très utilisé dans la preuve formelle.

Avoir effectué ce stage m'a permis d'avoir une meilleure compréhension des tables de hachage que j'avais étudié en cours d'algorithmique et en PFA (Programmation Fonctionnelle Avancée). De plus, cela m'a permis de redécouvrir les assistants de preuves, car lors de mon cursus en LDD1, j'ai suivi un cours qui visait à apprendre la rédaction de preuves en utilisant une partie de l'assistant de preuve LEAN [8]. Je trouve dommage qu'il n'y ait pas de continuité de ce cours en L2/L3, à l'exception du TER, car sans cette expérience, je n'aurais pas été en mesure de réaliser ce stage.

Grâce à ce stage, je suis confiant dans le fait que j'aurai de solides bases pour intégrer le *Master Parisien de Recherche en Informatique*. Cette expérience m'a confirmé que les méthodes formelles sont la voie que je souhaite suivre.

4 Remerciements

Pour conclure, j'aimerais remercier les personnes qui ont contribué au bon déroulement de mon stage et du TER. Tout d'abord, je tiens à remercier Sylvain Conchon pour m'avoir mis en contact avec Guillaume, que je remercie également pour m'avoir appris tant de choses tout au long de cette année et pour m'avoir accompagné une journée par semaine pendant un an. Je souhaite également exprimer ma gratitude envers Jean-Christophe Filliâtre pour m'avoir généreusement prêté des livres, ce qui a suscité mon intérêt pour la lecture scientifique. Enfin, je tiens à remercier tous les chercheurs du LMF pour leur accueil chaleureux.

Ces personnes ont joué un rôle essentiel dans mon parcours et ont contribué à rendre cette expérience enrichissante et gratifiante.

Références

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. *ITP*, 6172:83–98, 2010.

[Bak91] Henry G Baker. Shallow binding makes functional arrays fast. *ACM Sigplan Notices*, 26(8):145–147, 1991.

[BC15] Yves Bertot and Pierre Castéran. *Le Coq'Art (v8)*. Springer, 2015.

[BJM14] Thomas Braibant, Jacques-Henri Jourdan, and David Monniaux. Implementing and reasoning about hash-consed data structures in Coq. *Journal of Automated Reasoning*, 53(3):271–304, 2014.

[CF07] Sylvain Conchon and Jean-Christophe Filliâtre. A persistent union-find data structure. In *Proceedings of the 2007 on Workshop ML*, pages 37–46, 2007.

8. https://leanprover.github.io/