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 faite il possède un langage de programmation qui peut être extrait vers OCaml où directement interpréter. 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 donnée de haut niveau. Le but de ce stage est de se pencher sur les structures de données associatives en utilisant les tableaux persistants ajouter 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 extrait vers OCaml donc il y a peut d'intérêt à créé 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 seule structures de dictionnaires disponibles sont les FMapAVL qui sont difficile à prendre en main et peut performant. Une implémentation de table de hachage est disponible dans la bibliothèque standard de Coq, elle à 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 basés 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 naïf de table de hachage pour avoir une première base travaille. Pour la résolution de collision j'ai utiliser des seaux, c'est-à-dire que les case du tableaux sont composées de liste chainée, donc si deux éléments avec deux clé différentes sont attribué à la même case alors ils sont tout les deux ajouté 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 celle d'OCaml, c'est-à-dire de cacher les anciennes valeur d'une clé à chaque ajout. À cause de ce choix j'ai dût définir une nouvelle fonction `findall` qui donne toutes les valeurs associé 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'.
```

2.3.2 Tableaux dynamique

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

J'ai décomposé le problème en 3 fonctions :

— `rehash_bucket`: récupère les élément 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 externe récursive. Pour commencer, j'ai dû attribuer certaines propriétés aux deux autres fonctions, qui ont été admise initialement pour m'assurer de leur utilité par la suite. Une fois que cette étape de preuve a été réussie, j'ai dû prouver 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 dû ajouter des conditions dans le programme afin de vérifier si un élément se trouve effectivement dans le bon compartiment (bucket).

2.3.3 Tests réalisé

Pour pouvoir commencer les tests, j'ai ajouté les fonctions manquantes telles que `replace` ou `mem` [5]. Au départ, j'ai cherché à mettre en place 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 débordement. Les temps d'exécutions ne sont donc pas représentatifs des performance des dictionnaire.

Pour régler ce problème j'ai donc cherché une fonction memoïsante plus difficile à calculer. J'ai donc choisi de faire une fonction qui calcule les coefficient binomiaux à l'aide du triangle de pascal. Ma fonction a une complexité de $\mathcal{O}(n^2)$. Pour la memoïsation, j'ai comme type de clé un couple d'entier et comme valeur un entier. Il faut aussi crée une fonction de hachage qui éviterai le plus possible les collisions, j'ai choisi d'utiliser $h(k1, k2) = k1 + k2 \times 345$. Il a fallut aussi faire une version avec une paire de *positive* [6] car les arbres de Patricia vont utiliser ce type. J'ai utilisé la même fonction puis transformer l'entier machine en *positive*, cette transformation est de complexité $\mathcal{O}(\log(n))$ donc un peut couteuse. Il y a donc deux versions du tests: 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 tests avec les FMap de la bibliothèque standard de Coq [7].

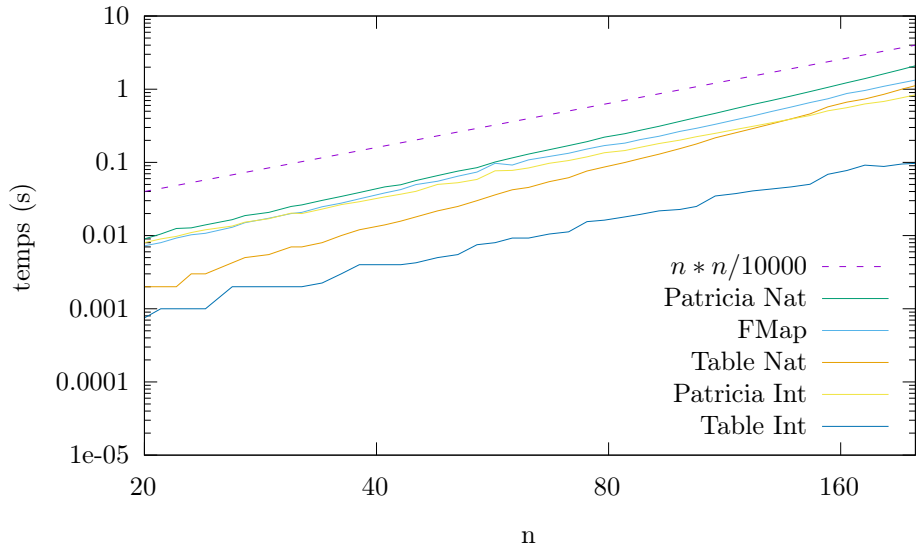

FIGURE 1 – Temps d'exécution de pascal_memo($2n$, n) dans un repère log-log

J'ai pût mesurer de très bonnes performances pour les tables de hachage sur les entiers Tableau-1. Table int et Patricia représente les deux structure crée avec une fonction de hachage gratuite.

5. Hashtable Ocaml. https://v2.ocaml.org/api/Hashtbl.html
6. Représentation d'un nombre en binaire avec une liste chainée
7. FMap. https://coq.inria.fr/library/Coq.FSets.FMapAVL.html

Contrairement à Table Nat et Patricia Nat qui utilise une fonction de hachage coûteuse.

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 croisé, 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 à déjà été rencontré (passage à la prochaine valeur). Les tests de performances ont été effectué avec deux types de valeur numérique : les entier machines et les *positive*. J'ai testé avec mon implémentation des arbres de Patricia, nos tables et les FMap de Coq. J'ai aussi utilisé les PositiveMap [8] de Coq qui sont aussi des arbres de Patricia.

n	Table Int	Patricia Int	FMap Int	Table Positive	Patricia Positive	PositiveMap
100	0.003s	0.005s	0.004s	0.003s	0.003s	0.002s
1000	0.021s	0.049s	0.044s	0.023s	0.035s	0.021s
1000	0.146s	0.241s	0.403s	0.191s	0.158s	0.202s
10000	1.243s	2.44s	3.87s	1.747s	1.794s	1.5s
100000	13.101s	29.101s	48.856s	18.251s	19.57s	16.969s

TABLE 1 – Vérification de la conjecture de syracuse de 1 à n

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

8. Patricia. 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 l'un des concepts clés de la preuve formelle. Les cours de programmation, tels que celui d'algorithmique en L2 et PFA, m'ont également donné de bonnes bases pour améliorer la qualité de mon code.

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 Remerciement

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 s'être occupé de moi 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 workshop on Workshop on ML*, pages 37–46, 2007.