Stage L3: Formalisation des dictionnaires en Coq

Valeran MAYTIE

Juillet 2023

1 Structure d'accueil

Mon stage se déroule au LMF [1] (Laboratoire de Méthodes Formelles) 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 générale

Coq est un assistant de preuve basé sur la théorie des 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 cet assistant de preuve en un outil de calcul formel. Un élément clé est la conception d'un langage de programmation dédié ainsi que des structures de données de haut niveau.

Plus généralement 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.

Le but de ce stage est de se pencher sur les structures de données associatives en utilisant entre autres les tableaux persistants ajoutés en 2010 à Coq.

2.2 Les enjeux

Les programmes vérifiés en Coq ont souvent pour but d'être extrait 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, on peut trouver deux structures de dictionnaires dans la bibliothèque standard de Coq : les *FMapAVL* [4] qui sont difficiles à prendre en main et peu performants et des arbres de Patricia [5] qui imposent que la clé soit un entier strictement positif. Une structure de données plus rapide serait bénéfique pour des opérations telles que la mémoïsation ou le partage maximal [BJM14].

Il serait également intéressant d'étudier si l'utilisation de structures impératives persistantes peut améliorer l'efficacité des programmes. Il est possible qu'elles utilisent beaucoup de mémoire, ce qui peut entraîner des temps d'allocation plus longs et augmenter le temps d'exécution du *garbage collector* (ramasse-miettes).

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

2.3 Travail effectué

Le premier travail était de comprendre comment les tableaux et les entiers machines sont implémentés en Coq en se référant à la littérature existante [AGST10]. L'implémentation des tableaux utilise la structure persistante de Henry G. Baker [Bak91]. Pour approfondir mes connaissances, j'ai aussi lu l'article 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 à base de tableaux persistants, j'ai formalisé des tables de hachage basées sur des arbres de Patricia afin d'avoir une idée des spécifications des fonctions de base. En me renseignant sur les différentes implémentations des tables de hachage, j'ai décidé de résoudre les collisions avec des seaux, c'est-à-dire que les feuilles de l'arbre sont composées de listes chainées, deux éléments avec des clés ayant la même valeur de hachage sont ajoutés dans une même liste.

Ensuite, j'ai défini les fonctions de base : `add`, `empty`, `remove` et `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 vérifié que les fonctions ont les spécifications attendues. Par exemple pour `add` j'ai prouvé les lemmes suivants :

```
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'.
```

J'ai écrit la plupart des fonctions en récursif terminal pour des raisons d'efficacité. Pour pouvoir faire des preuves, 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 persistants

Dans un langage impératif, une table de hachage efficace aurait été implémenté avec des tableaux plutôt que des arbres de Patricia. J'ai donc défini un autre type de table de hachage qui utilise les tableaux persistants de Coq :

```
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;
}.
```

Contrairement aux arbres de Patricia, il faut faire attention au calcul du modulo pour choisir la case du tableau [Knu98, §6.4]. Il faut aussi écrire une fonction `resize` dont la spécification est la suivante :

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

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 seau (complication pour `findall` et `resize`). Deux solutions étaient possibles : soit établir un invariant sur la structure de données, soit

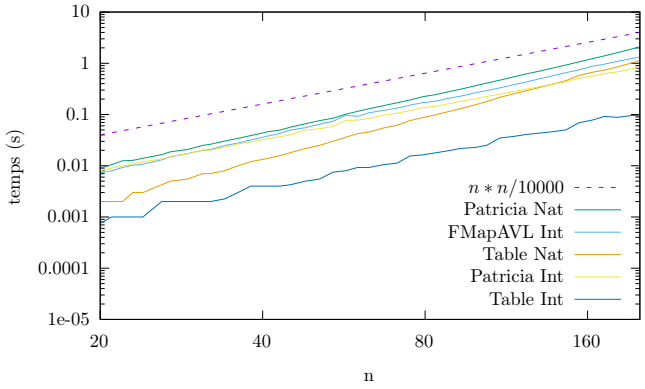

FIGURE 1 – Temps d'exécution de pascal_memo($2n$, n) dans un repère log-log

ajouter des conditions dans le programme pour gérer les cas impossibles dans les preuves. J'ai choisi la deuxième solution, car la présence d'un invariant aurait causé la création de termes de preuve très complexes à l'exécution. En effet, ils auraient conservé un historique des écritures dans la table ce qui aurait empêché le *garbage collector* de libérer la mémoire et aurait dégradé les performances.

Pour cela il a fallu que j'ajoute certaines conditions dans le programme afin de vérifier si un élément se trouve effectivement dans le bon seau. Sans cela les fonctions n'auraient pas vérifié leur spécification. Par exemple, pour la fonction `rehash_bucket` utilisée par `resize`, j'ai rajouté le test suivant :

```
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  (* teste 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].
```

Les preuves de correction ont été bien plus difficiles que dans le cas des arbres de Patricia. En effet, les tableaux de Coq utilisent les entiers machines sur lesquels il y a très peu de théorèmes, même pas un récurseur. Par ailleurs, pour une fonction comme `resize` il y a deux récursions imbriquées et de nombreux cas à traiter, la preuve de correction fait ainsi plus de 200 lignes de Coq.

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'entiers 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*[6] 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 coûteuse et l'autre avec une fonction de hachage gratuite.

6. Représentation d'un nombre en binaire avec une liste chainée.

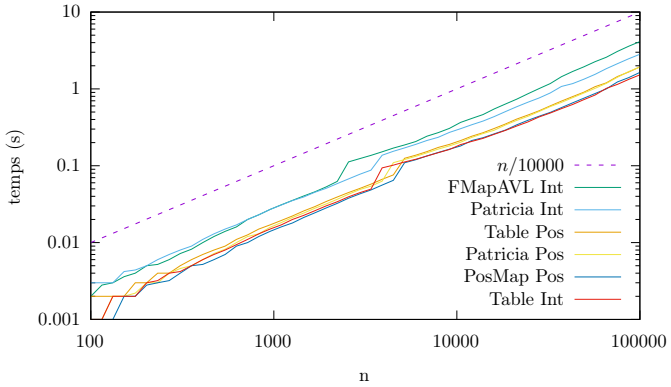

FIGURE 2 – Temps d'exécution de syracuse_test(n) dans un repère log-log

Les tests de performances ont été effectués avec deux types de clés : les couples d'entiers machines ("Int") et les couples d'entiers bâtons ("Nat"). J'ai testé mon implémentation des tables de hachage avec arbres de Patricia ("Patricia") et avec tableaux ("Table"). J'ai aussi utilisé les *FMap AVL* de Coq. Les résultats sont montrés dans la Figure 1. On y voit que les tableaux persistants rendent les tables de hachage 10 fois plus performantes que les arbres de Patricia. Par ailleurs, les *FMapAVL* sont 50% plus lent que les tables de hachage avec arbre de Patricia.

J'ai aussi écrit 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 ("Int") et les *positive* ("Pos"). J'ai testé avec mon implémentation des tables de hachage avec arbres de Patricia ("Patricia") et avec tableaux ("Table"). J'ai aussi utilisé les *FMap AVL* et les *PositiveMap* ("PosMap") de Coq. Les résultats sont donnés à la Figure 2.

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. Quand 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.

2.3.4 Conclusion

Pour conclure, j'ai fait deux implémentations de tables de hachage, une à base d'arbres de Patricia et une autre avec des tableaux persistants. Elles ont été toutes les deux vérifiées formellement. L'utilisabilité des spécifications a été testée sur la fonction de Pascal. Dans tous les tests les tables de hachage sont bien plus performantes que les *FMapAVL* de Coq. De plus, la version avec tableaux persistants est jusqu'à 10 fois plus rapide que les autres implémentations sur certains tests.

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. Il serait aussi intéressant de comprendre pourquoi les tableaux persistants n'apportent pas toujours un gain par rapport aux arbres de Patricia. Enfin, il faudra que je fasse une librairie Coq pour rendre les tables de hachage utilisables pour tous.

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 Castéran [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 important de la théorie des types.

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 surcouche en français de l'assistant de preuve LEAN [7]. 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. In *Interactive Theorem Proving*, volume 6172 of *LNCS*, pages 83–98. Springer, 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 *Workshop ML*, pages 37–46, 2007.

[Knu98] Donald E. Knuth. *The Art of Computer Programming Volume 3 Sorting and Searching*. Addison Wesley, 2nd edition, 1998.

7. https://leanprover.github.io/