Stage L3: Formalisation des dictionnaire en Coq

Valeran MAYTIE

Juillet 2023

1 Structure d'accueil

Notre stage se déroule au bâtiment 650 de l'Université Paris-Saclay, plus précisément 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, dont deux responsables : Sylvie Boldo (responsable permanente) et Claude Marché (responsable scientifique). Parmi eux se trouve Guillaume Melquiond, notre encadrant de stage. Ses travaux de recherche se situent à l'intersection des domaines de l'arithmétique informatique et de la preuve formelle.

2 Contexte scientifique

Le sujet du stage s'intitule *Formalisation des dictionnaire en Coq*.

2.1 Présentation Général

Le projet ERC Fresco [3] vie à transformer Coq en un outil rapide de calcul algébrique. Un élément clé est la conception d'un langage de programmation dédié ainsi que des structure de donnée de haut niveaux. Le but de ce stage est de ce pencher sur les structures de données associatives, c'est-à-dire les dictionnaires.

Bien sûr des personnes ont déjà formalisées des dictionnaire en Coq, mais elles sont toutes faite à l'aide d'arbre binaire, comme les arbres de Patricia (par exemple dans CompCert) ou des arbres AVL équilibré (dans le module FMap de la bibliothèque standard de Coq).

2.2 Les enjeux

1. LMF. https://lmf.cnrs.fr/
2. Inria. https://toccata.gitlabpages.inria.fr/toccata/index.fr.html
3. Fresco. https://fresco.gitlabpages.inria.fr/

2.3 Travail effectué

Le but du stage est d'explorer différentes implémentation de dictionnaire à base de tableau en Coq. Le premier travaille réalisé était de comprendre comment sont implémenter les tableau en Coq en lisant la littérature à se sujet. Pour cela nous avons chercher l'article qui parle de l'apparition de ces tableaux [AGST10]. Nous avons aussi pût trouver des informations sur les entier machine qui ont été implémenter en Coq en même temps, ils sont utilisé par les tableaux. En lisant ça j'ai très vite compris que les tableaux utilisées sont ceux de Baker. Pour compléter le cours au collègue de France de Xavier Leroy, je suis aller lire l'article de Baker [Bak91] et celui de Sylvain Conchon, Jean-Christophe Filliâtre [CF07].

2.3.1 Implémentations

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

Nous avons dû commencer à utiliser les tableaux de Coq et à comprendre leur interface d'utilisation. Toutes les fonctions de base, telles que `get`, `set`, `length` et `copy`, sont disponibles. De plus, il existe une fonction `make` qui permet de créer un tableau en spécifiant un type, une taille et une valeur par défaut pour éviter les comportements indéfinis.

Ainsi, nous avons défini le type des tables de la manière suivante :

```
Inductive bucket : Set :=
  | Empty : bucket
  | Cons : int → A → B → bucket → bucket.

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

Ensuite, nous avons défini les fonctions de base : `add`, `empty`, `find` et `findall`. Nous avons de choisir de faire comme les tables de hachage en Ocaml, c'est à dire de recouvrir les anciennes valeur d'une clé à chaque ajout d'où l'utilisation du `findall`. Nous avons donc fait notre première preuve sur les tableaux en spécifiant ces fonctions.

```
Parameter find_spec:
  forall B (ht: t B) key,
  find ht key = List.hd_error (find_all ht key).

Parameter hempty:
  forall B k s, find (create B s) k = None.

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

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

Pour effectuer ces preuves, nous avons dû commencer par créer une version non récursive terminale de la fonction `findall` afin de pouvoir effectuer des preuves par calcul dans Coq, ce qui facilite grandement les démonstrations. Ensuite, nous avons rencontré une première difficulté : un utilisateur pourrait fournir n'importe quelle table en tant que paramètre, et nous n'avons donc aucune information

sur la valeur par défaut de la table. Deux solutions étaient possibles : soit établir un invariant sur notre structure de données, soit ajouter des conditions dans le programme pour gérer les cas impossibles dans les preuves. Nous avons 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 (il aurait souvent fallu passer par des représentations de nombres peu performantes dans les preuves).

Pour avoir une structure performante, il faut transformer les tableaux en tableaux dynamique. Nous avons décomposé le problème en 3 fonctions :

— `rehash_bucket` : replace 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

Nous avons 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 c'est la première grande preuve que nous avons réalisée entièrement par nous-mêmes. Pour commencer, nous avons dû attribuer certaines propriétés aux deux autres fonctions, qui ont été acceptées initialement pour nous assurer de leur utilité par la suite. Une fois que cette étape de preuve a été réussie, nous avons dû prouver toutes les assertions liées à la fonction en question, parfois sans disposer de suffisamment d'hypothèses. Pour mener à bien cette preuve, nous avons également dû ajouter des conditions dans le programme afin de vérifier si un élément se trouve effectivement dans le bon compartiment (bucket).

2.3.2 Tests

Pour pouvoir commencer les tests, nous avons ajouté les fonctions manquantes telles que `replace` ou `mem` (Hashtable d'Ocaml [4]). Au départ, nous avons 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 dépassements de capacité.

3 Retour d'expérience

Ce stage à été une bonne opportunité pour apprendre de nouvelles choses et en approfondir certaines.

4 Remercient

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.

[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.

4. Hashtable Ocaml. https://v2.ocaml.org/api/Hashtbl.html