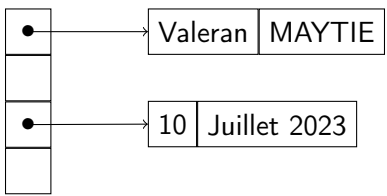
Formalisation des dictionnaires en Coq



Présentation de Coq

Coq est un assistant de preuve basé sur la théorie des types.
De ce fait il possède un langage de programmation appelé *Gallina*.
Ce langage peut être extrait ou interprété.

On crée des structures de données pour écrire des algorithmes.
Il est donc intéressant de chercher à les optimiser.

Assistant de preuve Coq

```
Lemma pascal_zero :
  forall n m, n < m → pascal n m = 0.
Proof.
  induction n; simpl.
  + destruct m; easy.
  + destruct m. easy. intros H.
    apply Nat.succ_lt_mono in H.
    rewrite 2!IHn. easy.
    apply Nat.lt_lt_succ_r; easy.
Qed.

Theorem pascal_one :
  forall n, pascal n n = 1.
Proof.
  induction n. reflexivity.
```

```
1 subgoal

n : nat
IHn : forall m : nat, n < m → pascal n m = 0
m : nat
H : n < m

============================ (1 / 1)

pascal n m + pascal n (S m) = 0
```

Dictionnaires en Coq

Il y a déjà deux structures de dictionnaires formalisées en Coq.

- FMapAVL, basé sur des arbres binaires équilibrés
- FMapPositive, basé sur des arbres PATRICIA (utilisation des entiers *Positive*).

Il n'y a pas de table de hachage.

Objectifs

Implantation et vérification de table de hachage en Coq

- Arbres PATRICIA
- Tableaux persistants

Évaluer les perfomances

1. Introduction
2. Préliminaires
   - Coq
   - Arbres binaires de recherche
   - Arbres PATRICIA
   - Tableaux de Baker
   - Tables de hachage
3. Tables de hachage en Coq
4. Tableaux persistants
5. Tests
6. Conclusion

Langage Coq

Langage très proche de Caml, mais "pur" (pas d'effet de bord)

- Types inductifs et pattern matching
- Fonctions récursives (Fixpoint, termine forcément)

```
Inductive nat : Set :=
| O : nat
| S : nat → nat.
```

```
Inductive positive : Set :=
| xI : positive → positive
| xO : positive → positive
| xH : positive.
```

Exemple de programme Coq

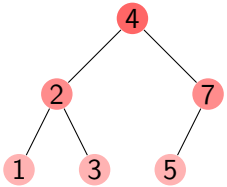
Calcul des coefficients binomiaux à l'aide du triangle de Pascal :

$$\binom{n}{k} = \binom{n-1}{k-1} + \binom{n-1}{k}$$

```
Fixpoint pascal (n m: nat) : int :=
  match n, m with
  | 0, 0 ⇒ 1
  | 0, _ ⇒ 0
  | _, 0 ⇒ 1
  | S n', S m' ⇒ pascal n' m' + pascal n' m
  end.

Compute pascal 10 5.
  = 252
      : int
```

Arbres binaires de recherche



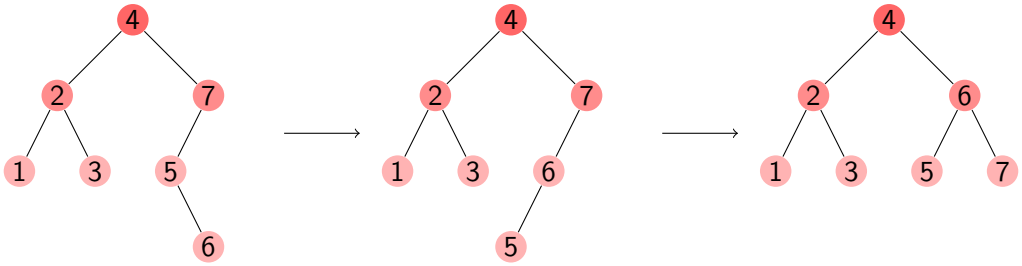

```
type 'a tree =
| Empty
| Node of 'a tree * 'a * 'a tree
```

Pour un arbre binaire de recherche équilibré on ajoute des invariants sur les nœuds :

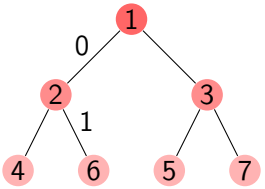
1. toutes les valeurs des descendants à droite sont plus petites
2. toutes les valeurs des descendants à gauche sont plus grandes
3. la différence entre les profondeurs est au plus 1 (AVL)

Insertion et équilibrage

L'équilibrage se fait par des rotations



Arbres PATRICIA

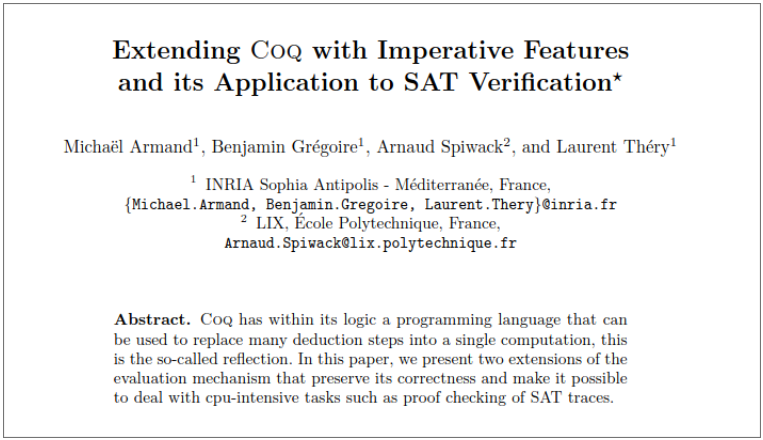


Arbre binaire pour des identifiants entiers.

```
Fixpoint get (p: positive) (t: tree) :=
  match p, t with
  | _,     Empty       ⇒ None
  | xH,    Node _ e _  ⇒ e
  | xO p', Node l _ _  ⇒ get p' l
  | xI p', Node _ _ r  ⇒ get p' r
  end.
```

Par exemple, la clé $6 = 110_2$ représente le chemin : gauche puis droite.

Tableaux persistants de Coq



Extending Coq with Imperative Features and its Application to SAT Verification*

Michaël Armand[1], Benjamin Grégoire[1], Arnaud Spiwack[2], and Laurent Théry[1]

[1] INRIA Sophia Antipolis - Méditerranée, France,
{Michael.Armand, Benjamin.Gregoire, Laurent.Thery}@inria.fr
[2] LIX, École Polytechnique, France,
Arnaud.Spiwack@lix.polytechnique.fr

Abstract. Coq has within its logic a programming language that can be used to replace many deduction steps into a single computation, this is the so-called reflection. In this paper, we present two extensions of the evaluation mechanism that preserve its correctness and make it possible to deal with cpu-intensive tasks such as proof checking of SAT traces.

Spécification des tableaux

```
make : int → A → array A
length : array A → int
get : array A → int → A
set : array A → int A → array A

Axiom get_set_same :
  forall t i a,
    (i <? length t) = true → t.[i ← a].[i] = a.

Axiom get_set_other :
  forall t i j a,
    i <> j → t.[i ← a].[j] = t.[j].
```

Tableaux de Baker

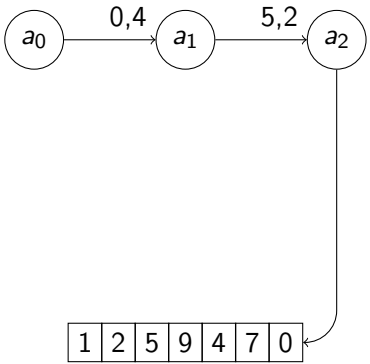


Tableau en apparence persistant,
il est implanté à l'aide de traits impératifs.

```
type 'a t = ('a kind) ref
and 'a kind =
| Array of 'a array * 'a
| Updated of int * 'a * 'a t
```

Tableaux de Baker

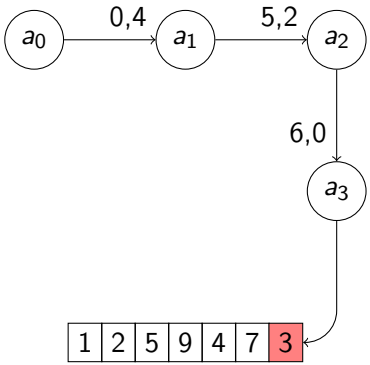
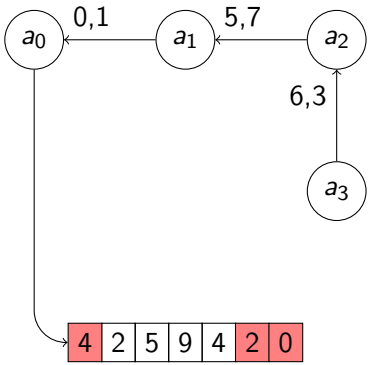


Tableau en apparence persistant,
il est implanté à l'aide de traits impératifs.

```
type 'a t = ('a kind) ref
and 'a kind =
| Array of 'a array * 'a
| Updated of int * 'a * 'a t
```

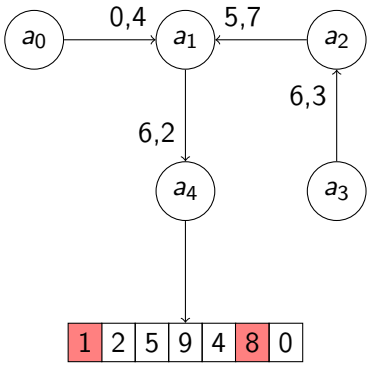
Tableaux de Baker, *reracinage*



Le dernier accès en écriture ou en lecture pointe toujours vers le tableau.

$9 = a_0\,.\,[3]$

Tableaux de Baker, *reracinage*



Le dernier accès en écriture ou en lecture pointe toujours vers le tableau.

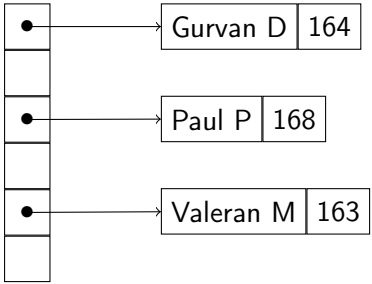
$a_4 = a_1\,.\,[5\ \texttt{<-}\ 8]$

Tables de hachage

Table associative clé-valeur

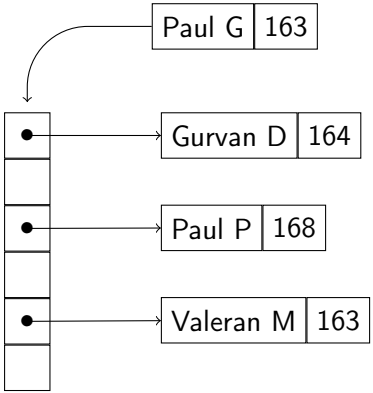
Utilisation d'une fonction de hachage pour transformer les clés en entiers

Fonction de hachage : Deux premières lettres



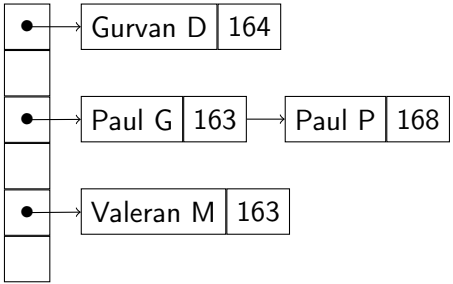
Collisions

Une collision se produit quand deux *hashs* pointent vers la même case.

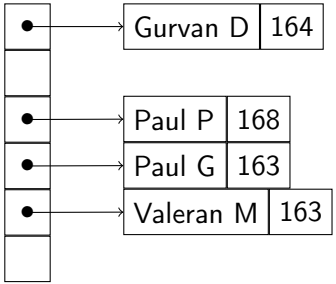


Résolution des collisions

Résolution avec liste chainée :



Résolution avec adressage ouvert



1. Introduction
2. Préliminaires
3. Tables de hachage en Coq
   - Implantation
   - Preuves
4. Tableaux persistants
5. Tests
6. Conclusion

Première table de hachage : arbres PATRICIA avec seaux

Utilisée pour faire des tests et se familiariser avec Coq.

```
Inductive bucket : Set :=
| B_Empty : bucket
| B_Cons (hash: positive) (key: A) (value: B) (next: bucket) : bucket.

Inductive tree : Type :=
| Empty: tree
| Node:  tree → bucket → tree → tree.
```

Fonctions de base

Même propriété que les tables de hachage de OCaml : plusieurs fois la même clé.

Interface : `create`, `add`, `remove`, `find` et `find_all`.

```
Lemma create_spec: forall k s,
  find_all (create s) k = [].

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

Lemma add_diff: forall h k k' v,
  k' <> k → find_all (add h k v) k' = find_all h k'.
```

Spécification du type des clés

```
Module Type Hash_type.
  Parameter A: Set.
  Parameter eq: A → A → bool.
  Parameter eq_spec: forall x y : A, reflect (x = y) (eq x y).
  Parameter hash: A → positive.
End Hash_type.
```

Aperçu des preuves

Certaines fonctions sont récursives terminales,
il y a donc des problèmes pour faire une démonstration par récurrence.

Solution :

1. Écriture des fonctions en non terminal
2. Preuves qu'elles ont les mêmes résultats

Exemple pour `find_all` :

```
Lemma find_all_rec_correct:
  forall (l: bucket) (h: int) (key: A) acc,
  find_all_rec l h key acc = rev acc ++ find_all_rec' l h key.
```

1. Introduction
2. Préliminaires
3. Tables de hachage en Coq
4. Tableaux persistants
   - Implantation
   - Redimensionnement
   - Preuves
5. Tests
6. Conclusion

Tables de hachage avec tableau

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

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

Redimensionnement

Redimensionnement des tableaux pour éviter le plus possible les collisions.

```
Definition resize_heuristic (h: t) : bool :=
  (length h << 1 <=? size h) && negb (length h =? PArray.max_length).
```

Redimensionnement

```
Definition rehash_bucket (new_tab  old_tab: t)
                         (new_size old_size i: int) : t :=
  fold_right (fun hash key value a ⇒
    let h_b := key_index new_size hash in
    (* copie dans le nouveau tableau *)
    a.[h_b ← Cons hash key value a.[h_b]])
  new_tab old_tab.[i].

Lemma rehash_bucket_correct:
  ... → find_all (rehash_bucket nt lt ...) k = find_all lt k.
```

⚠ Impossible à prouver !

Complication : invariants structurels

Une table pourrait venir de n'importe où donc potentiellement mal formée.

Deux choix possibles :

1. Invariant structurel $\Rightarrow$ termes de preuves stockés dans les tables de hachage
2. Ajout de tests pour détecter les cas où la table est mal formée.

Résolution : conditions supplémentaires

```
Definition rehash_bucket (new_tab  old_tab: t)
                         (new_size old_size i: int) : t :=
  fold_right (fun hash key value a ⇒
    (* test si l'element est dans le bon seau *)
    if i =? key_index old_size hash then
      let h_b := key_index new_size hash in
      (* copie dans le nouveau tableau *)
      a.[h_b ← Cons hash key value a.[h_b]]
    else a)
  new_tab old_tab.[i].
```

Effort de preuve

Preuves de correction bien plus difficiles que celles des arbres :

- Plusieurs récurrences imbriquées (seaux et tableaux)
- Beaucoup de cas à gérer (conditions ajoutées)
- Pas mal d'arithmétique avec les entiers machines (modulo)

Exemple : environ 200 lignes de preuve pour `resize`.

1. Introduction
2. Préliminaires
3. Tables de hachage en Coq
4. Tableaux persistants
5. Tests
   - Triangle de Pascal
   - Conjecture de Syracuse
6. Conclusion

Triangle de Pascal

```
Fixpoint pascal (n m: nat) : int :=
  match n, m with
  | 0, 0 ⇒ 1
  | 0, _ ⇒ 0
  | _, 0 ⇒ 1
  | S n', S m' ⇒ pascal n' m' + pascal n' m
  end.
```

Memoïsation du triangle de Pascal

```
Fixpoint pascal_memo' (n m: nat) (h: H.t int) : (int * H.t int) :=
  match H.find h (N n m) with
  | Some v ⇒ (v, h)
  | None   ⇒ match n, m with
             | 0, 0 ⇒ (1, h)
             | 0, _ ⇒ (0, h)
             | _, 0 ⇒ (1, h)
             | S n', S m' ⇒
                 let (v1, h1) := pascal_memo' n' m' h in
                 let (v2, h2) := pascal_memo' n' m h1 in
                 let r := v1 + v2 in
                 (r, H.add h2 (N n m) r)
           end
  end.
```

Comparaison de quatre dictionnaires

- Arbres AVL
- Arbres PATRICIA (clés = entiers)
- Tables de hachage avec arbres PATRICIA
- Tables de hachage avec tableaux persistants

Résultats

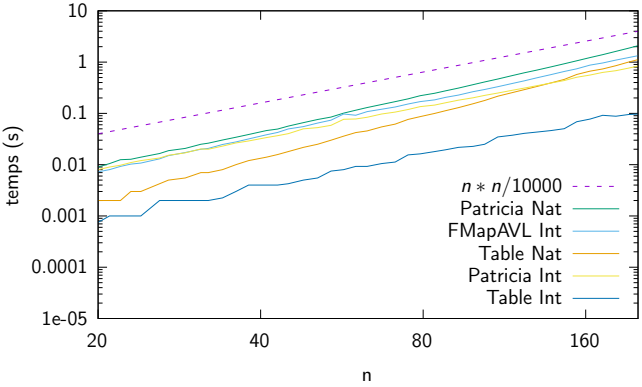


Figure: Temps d'exécution de pascal_memo($2n$, n) dans un repère log-log

Suite de Syracuse

$$u_{n+1} = \begin{cases} u_n/2 & \text{si } u_n \text{ est pair} \\ 3 \times u_n + 1 & \text{sinon} \end{cases}$$

On veut tester si cette suite finit toujours par boucler avec les valeurs :
1, 4, 2, 1, 4, 2, 1, ...

Implantation en OCaml avec break et continue

```
let h = H.create 16 in
for i = 2 to n do
  if H.mem h i then (H.remove h i; continue);
  let j = ref i in
  while true do
    j := syracuse !j;
    if !j < i then break;
    match H.find_opt h !j with
    | Some i' -> if i = i' then return false else break
    | None    -> H.add h !j i
  done
done;
true
```

Résultats

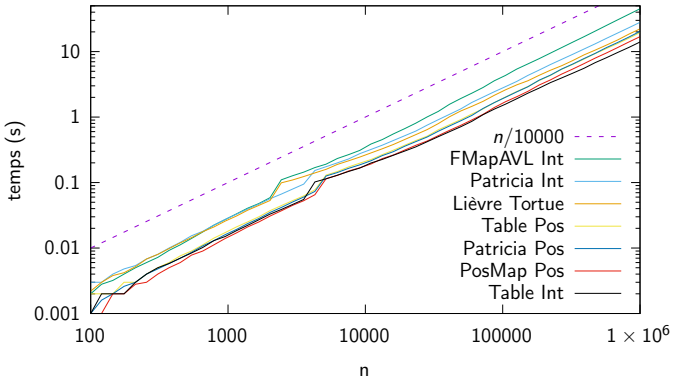


Figure: Temps d'exécution de syracuse_test(n) dans un repère log-log

1. Introduction
2. Préliminaires
3. Tables de hachage en Coq
4. Tableaux persistants
5. Tests
6. Conclusion

Conclusion

Contribution : deux structures de tables de hachage vérifiées formellement :

- arbre PATRICIA
- tableau persistant

Conclusion

Contribution : deux structures de tables de hachage vérifiées formellement :

- arbre PATRICIA
- tableau persistant

Perspectives :

1. Faire plus de tests (autres fonctions)
2. Comprendre pourquoi il n'y a pas toujours de gain par rapport aux arbres PATRICIA
3. Rendre les tables de hachage utilisables par tous

Conclusion

Contribution : deux structures de tables de hachage vérifiées formellement :

- arbre PATRICIA
- tableau persistant

Perspectives :

1. Faire plus de tests (autres fonctions)
2. Comprendre pourquoi il n'y a pas toujours de gain par rapport aux arbres PATRICIA
3. Rendre les tables de hachage utilisables par tous