Union-Find Persistant

Sylvain Conchon[1] et Jean-Christophe Filliâtre[1]

1: Laboratoire de Recherche en Informatique – CNRS
Bâtiment 490, Université Paris Sud
91405 ORSAY Cedex, France
{conchon,filliatr}@lri.fr

Résumé

Le problème des classes disjointes, connu sous le nom de *union-find*, consiste à maintenir dans une structure de données une partition d'un ensemble fini. Cette structure fournit deux opérations : une fonction *find* déterminant la classe d'un élément et une fonction *union* réunissant deux classes. Une solution optimale et impérative, due à Tarjan, est connue depuis longtemps.

Cependant, le caractère impératif de cette structure de données devient gênant lorsqu'elle est utilisée dans un contexte où s'effectuent des retours en arrière (*backtracking*). Nous présentons dans cet article une version persistante de *union-find* dont la complexité est comparable à celle de la solution impérative. Pour obtenir cette efficacité, notre solution utilise massivement des traits impératifs. C'est pourquoi nous présentons également une preuve formelle de correction, pour s'assurer notamment du caractère persistant de cette solution.

1. Introduction

Le problème des classes disjointes, connu sous le nom de *union-find*, consiste à maintenir dans une structure de données une partition d'un ensemble fini. Sans perte de généralité, on peut supposer qu'il s'agit de l'ensemble des n entiers $\{0, 1, \ldots, n-1\}$. La figure 1 donne la signature[1] traditionnelle d'une telle structure. On y trouve un type abstrait `t` représentant une partition et trois opérations. L'opération `create` n construit une nouvelle partition où chaque élément forme une classe à lui tout seul. L'opération `find` détermine la classe d'un élément, sous la forme d'un entier considéré comme le représentant de cette classe. Enfin l'opération `union` réunit deux classes de la partition, la structure de données étant modifiée en place.

Une solution optimale, due à Tarjan, est connue depuis longtemps [13] et est un classique de la littérature algorithmique (par exemple [7] chapitre 22). Le code de cette solution est donné en annexe. L'idée principale est de lier entre eux les éléments d'une même classe. Pour cela, un tableau `father` lie chaque entier i à un autre entier de la même classe. Dans chaque classe, ces liaisons forment un graphe acyclique où tous les chemins mènent au représentant, qui est le seul élément lié à lui-même. La figure 3 illustre une situation où l'ensemble $\{0, 1, \ldots, 7\}$ est partitionné en deux classes dont les représentants sont respectivement 3 et 4. L'opération `find` se contente de suivre les liaisons jusqu'à trouver le représentant. L'opération `union` commence par trouver les représentants des deux éléments, puis lie l'un des deux représentants à l'autre. Afin d'atteindre des performances optimales, la solution de Tarjan apporte deux améliorations. La première consiste à *compresser les chemins* pendant la recherche effectuée par `find` : cela consiste à lier directement au représentant tous les éléments trouvés sur le chemin parcouru pour le trouver. La seconde consiste à maintenir pour chaque représentant un

[1] Cet article est illustré par du code écrit dans le langage Objective Caml [2], mais s'adapte aisément dans n'importe quel autre langage.

```
module type ImperativeUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → unit
end
```

FIG. 1 – Structure *union-find* impérative

```
module type PersistentUnionFind = sig
  type t
  val create : int → t
  val find : t → int → int
  val union : t → int → int → t
end
```

FIG. 2 – Structure *union-find* persistante

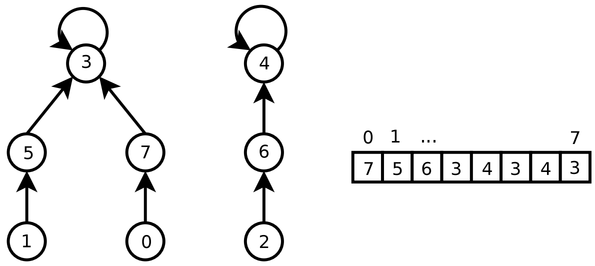

FIG. 3 – Une partition en deux classes de $\{0, 1, \ldots, 7\}$

niveau qui approxime la longueur du plus long chemin jusqu'à celui-ci, et à choisir comme représentant d'une union celui de plus grand niveau.

Bien que cette solution soit optimale, son caractère impératif devient gênant lorsqu'elle est utilisée dans un contexte où s'effectuent des retours en arrière. Un exemple significatif est celui des procédures de décision, où le traitement de la structure booléenne procède essentiellement par *backtracking*. Or le traitement de l'égalité dans ces outils est très souvent basé sur une structure *union-find* [10, 12]. Dès lors, il est nécessaire de pouvoir revenir à des versions antérieures de cette structure. Il est clair que la solution de Tarjan, avec sa compression de chemins et le maintien des niveaux, ne se prête pas facilement à ces retours en arrière.

Une solution simple au problème du retour en arrière consiste à utiliser une structure de données *persistante*, c'est-à-dire où l'opération `union` renvoie une nouvelle partition et laisse la précédente *inchangée*. La signature d'une telle structure persistante est donnée figure 2. Une manière simple de réaliser une telle signature consiste à utiliser des structures de données purement applicatives. Malheureusement, de telles solutions restent très loin des performances de la version impérative de Tarjan, en particulier parce que la signature de l'opération `find` ne permet pas la compression de chemins (il faudrait pour cela que `find` renvoie une nouvelle partition, ce qui obligerait le programmeur à modifier son code client). Pour atteindre de meilleures performances, nous allons donc nous tourner vers des structures utilisant des effets de bord mais tout en restant *persistantes*, c'est-à-dire *observationnellement immuables*. Notre contribution consiste en effet en une version persistante de l'algorithme de Tarjan utilisant une structure de données inspirée des tableaux persistants de Baker.

Cet article est organisé de la façon suivante. La section 2 présente une structure de données persistante pour le problème *union-find* dont l'efficacité est comparable à celle de sa version impérative. La section 3 présente une analyse des performances de notre solution, comparée à plusieurs autres solutions. Enfin, la section 4 détaille la preuve formelle dans l'assistant de preuve Coq de la correction de cette structure de données persistante.

2. Une solution à la fois très simple et très efficace

La solution au problème de l'*union-find* persistant que nous proposons est à la fois très simple et très efficace. Elle consiste à rester aussi proche que possible de l'algorithme original de Tarjan,

```
module type PersistentArray = sig
  type α t
  val init : int → (int → α) → α t
  val get : α t → int → α
  val set : α t → int → α → α t
end
```

Fig. 4 – Signature minimale des tableaux persistants

avec notamment sa compression de chemin, mais en substituant à l'usage de tableaux modifiés en place des *tableaux persistants*. Un tableau persistant est une structure de données offrant les mêmes fonctionnalités qu'un tableau usuel, à savoir la manipulation d'éléments indicés de 0 à $n-1$ avec des opérations d'accès et de mise à jour à très bas coût, mais où l'opération de mise à jour construit un nouveau tableau persistant et laisse le précédent inchangé. Une signature minimale pour des tableaux persistants polymorphes est donnée figure 4. Bien entendu, on peut facilement réaliser cette signature à l'aide de dictionnaires codés de manière purement applicative, tels que les arbres binaires de recherche de la bibliothèque standard d'Ocaml. Mais comme le montrent les tests de performance (voir section 3), le coût logarithmique des opérations d'accès et de mise à jour d'une telle structure est trop important. Heureusement, il est possible d'atteindre de bien meilleures performances pour les tableaux persistants, comme nous le montrerons plus loin (sections 2.2–2.4). En attendant, nous allons présenter une version persistante de l'algorithme de Tarjan, indépendamment de la réalisation des tableaux persistants.

2.1. Une version persistante de l'algorithme de Tarjan

Pour rester indépendants de la structure de données des tableaux persistants, nous introduisons naturellement un module paramétré par cette structure de données, *i.e.* un foncteur :

```
module Make(A : PersistentArray) : PersistentUnionFind = struct
```

Comme dans sa version impérative, la structure *union-find* est une paire de tableaux (`rank` qui contient les niveaux des représentants et `father` qui contient les liaisons) mais ce sont ici des tableaux persistants :

```
type t = { rank: int A.t; mutable father: int A.t }
```

Le caractère modifiable du second champ sera exploité pour réaliser la compression de chemin. La création d'une nouvelle structure *union-find* où chaque élément forme une classe à lui tout seul ne pose pas de problème :

```
let create n = { rank = A.init n (fun _ → 0); father = A.init n (fun i → i) }
```

Pour réaliser la compression de chemin qui est au cœur de l'algorithme de Tarjan, la fonction `find` doit effectuer des modifications des liaisons au fur et à mesure de sa remontée, une fois le représentant trouvé. Ici, les liaisons sont représentées par un tableau persistant et on doit donc construire un *nouveau* tableau lors de cette remontée. Pour cela on commence par définir une première fonction `find_aux` prenant en argument le tableau des liaisons `f` et l'élément `i` dont on cherche le représentant, et renvoyant le nouveau tableau et le représentant de `i` :

```
let rec find_aux f i =
  let fi = A.get f i in
  if fi == i then
    f, i
  else
    let f, r = find_aux f fi in
    let f = A.set f i r in
    f, r
```

Si ce n'est qu'elle opère sur un tableau persistant, cette fonction est identique à la fonction `find_aux` de la version impérative. On peut alors définir une fonction `find` qui va appeler la fonction `find_aux` puis *modifier* le champ `father` de la structure de données par un effet de bord pour entériner les changements survenus sur le tableau des liaisons :

```
let find h x =
  let f,rx = find_aux h.father x in h.father ← f; rx
```

Comme on le constate, cette fonction a bien le type escompté, à savoir qu'elle ne renvoie qu'un entier, mais elle effectue bien la compression de chemin. Comme dans la version impérative de `find`, la structure de données a été modifiée par effet de bord mais reste observationnellement inchangée : le représentant de chaque élément est toujours le même.

Pour réaliser la fonction `union`, on suit là encore le code de la version impérative, mais on renvoie maintenant une nouvelle structure de données, à savoir un nouvel enregistrement de type `t`. Celui-ci contient les nouvelles versions des tableaux `rank` et `father`, le cas échéant.

```
let union h x y =
  let rx = find h x in
  let ry = find h y in
  if rx != ry then begin
    let rxc = A.get h.rank rx in
    let ryc = A.get h.rank ry in
    if rxc > ryc then
      { h with father = A.set h.father ry rx }
    else if rxc < ryc then
      { h with father = A.set h.father rx ry }
    else
      { rank = A.set h.rank rx (rxc + 1);
        father = A.set h.father ry rx }
  end else
    h
```

On obtient au final un code qui n'est pas vraiment plus long que celui de sa contrepartie impérative. Il nous reste le plus difficile : fournir une réalisation efficace des tableaux persistants en argument à ce foncteur.

2.2. Tableaux persistants

Une solution efficace au problème des tableaux persistants est en fait connue depuis longtemps. Il semble qu'elle soit due à H. Baker [9, 3], qui l'utilisait pour représenter efficacement les environnements dans des clôtures Lisp. L'idée de base consiste à utiliser un tableau usuel[2] pour la dernière version du

[2] À partir de maintenant, on utilise le terme « tableau » pour un tableau au sens usuel, et donc modifiable en place, et le terme « tableau persistant » sinon.

tableau persistant, et des indirections pour les versions antérieures. On introduit pour cela les deux types mutuellement récursifs suivants :

```
type α t = α data ref
and α data =
  | Arr of α array
  | Diff of int × α × α t
```

Le type α `t` est celui des tableaux persistants. Il s'agit d'une référence vers une donnée du type α `data` qui indique sa nature : soit une valeur immédiate `Arr` a avec a un tableau, soit une indirection `Diff`(i, v, t) représentant un tableau persistant identique en tout point au tableau persistant t hormis à l'indice i où l'on trouve la valeur v. La présence de la référence peut sembler superflue en premier abord, mais elle est cruciale. La création d'un nouveau tableau persistant est immédiate :

```
let init n f = ref (Arr (Array.init n f))
```

La fonction d'accès `get` n'est guère plus complexe. Soit le tableau persistant est immédiatement un tableau, soit il faut consulter l'indirection et si nécessaire poursuivre l'accès récursivement :

```
let rec get t i = match !t with
  | Arr a → a.(i)
  | Diff (j, v, t') → if i == j then v else get t' i
```

Toute la subtilité tient dans la fonction `set`. Comme nous l'avons dit plus haut, l'idée est de conserver les performances d'un tableau usuel pour la dernière version du tableau persistant, quitte à dégrader celles des versions antérieures. Lorsque l'on effectue une mise à jour dans le tableau persistant `t`, deux cas se présentent :

- soit `t` pointe sur une valeur du type `Arr` a ; dans ce cas, on va le « dérouter » pour qu'il pointe maintenant sur une indirection (ce qui est possible puisqu'il s'agit d'une référence et non pas directement d'une valeur du type α `data`) et renvoyer une nouvelle référence `res` pointant sur le tableau a, qui aura été modifié en place.
- soit `t` est déjà une indirection *i.e.* pointe sur une valeur du type `Diff` ; dans ce cas on se contente simplement de créer une nouvelle indirection.

Cela se traduit par le code suivant :

```
let set t i v = match !t with
  | Arr a as n →
      let old = a.(i) in
      a.(i) ← v;
      let res = ref n in
      t := Diff (i, old, res);
      res
  | Diff _ →
      ref (Diff (i, v, t))
```

Comme on le constate, une valeur de la forme `Arr` a ne peut être créée que par la fonction `init`. En conséquence, une succession d'opérations de mise à jour à partir d'un premier tableau persistant construit avec `create` n'aura alloué qu'un seul tableau et un espace supplémentaire proportionnel au nombre d'appels à `set` (car il est clair que `set` a une complexité $O(1)$ en temps et en espace). Si on considère la série de déclarations suivantes définissant quatre tableaux persistants `a0`, `a1`, `a2` et `a3`

```
let a0 = create 7 0
let a1 = set a0 1 7
```

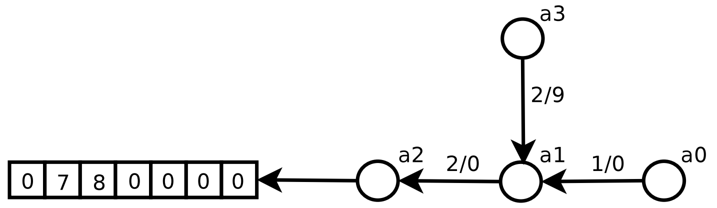

FIG. 5 – Illustration des tableaux persistants (1/2)

```
let a2 = set a1 2 8
let a3 = set a1 2 9
```

alors la situation à l'issue de ces déclarations est illustrée figure 5, où chaque référence est représentée par un cercle et un bloc `Diff` par une arête étiquetée. D'une manière générale, on a la propriété suivante : le graphe des références de type α `t` est acyclique et de chacune d'entre elles il existe un unique chemin menant à la valeur `Arr`.

De tels tableaux persistants se comportent bien lorsque l'on accède toujours à la dernière version mais les performances se dégradent considérablement dès que l'on accède à des versions antérieures. En effet, une succession de mises à jour va créer une chaîne de constructeurs `Diff` et chaque accès à partir de cette chaîne aura un coût proportionnel à sa longueur *i.e.* au nombre d'opérations de mise à jour effectuées sur ce tableau et ses successeurs. C'est évidemment dramatique dans un contexte où l'on effectue des retours en arrière. Or c'est justement pour cela que l'on cherchait à construire des tableaux persistants. Heureusement, il existe une façon simple d'améliorer substantiellement cette première version.

2.3. Une amélioration substantielle

Pour remédier à la dégradation des performances lorsque l'on accède aux versions antérieures des tableaux persistants, H. Baker propose une amélioration toute simple : dès que l'on accède à un tableau persistant qui n'est pas immédiat, on commence par retourner la chaîne de pointeurs menant au tableau `Arr`, de manière à amener celui-ci à l'endroit où l'on cherche à accéder. Cette opération appelée *rerooting* par Baker peut être codée par la fonction `reroot` suivante, qui prend en argument un tableau persistant et ne renvoie aucune valeur ; elle se contente de modifier la structure des pointeurs, sans affecter le contenu des tableaux persistants.

```
let rec reroot t = match !t with
  | Arr _ → ()
  | Diff (i, v, t') →
      reroot t';
      begin match !t' with
        | Arr a as n →
            let v' = a.(i) in
            a.(i) ← v;
            t := n;
            t' := Diff (i, v', t)
        | Diff _ → assert false
      end
```

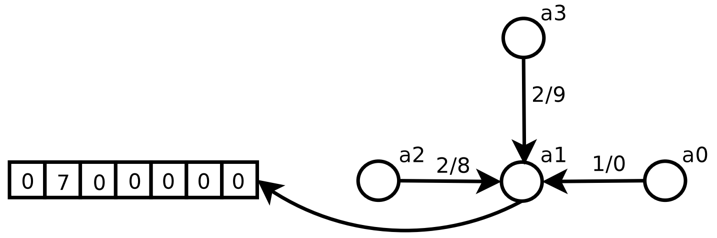

FIG. 6 – Illustration des tableaux persistants (2/2)

À l'issue de l'exécution de cette fonction, on a la propriété que `t` pointe maintenant sur une valeur de la forme `Arr`. On peut donc modifier la fonction d'accès de manière à appeler `reroot` dès que le tableau est de la forme `Diff` :

```
let rec get t i = match !t with
  | Arr a →
      a.(i)
  | Diff _ →
      reroot t;
      begin match !t with Arr a → a.(i) | Diff _ → assert false end
```

On peut modifier de même la fonction `set` :

```
let set t i v =
  reroot t;
  match !t with
  | Arr a as n → ... comme auparavant ...
  | Diff _ → assert false
```

Si on reprend l'exemple de la figure 5, et que l'on cherche à accéder à un élément de `a1`, celui-ci va d'abord subir l'opération `reroot`. Ceci va amener `a1` à pointer maintenant sur le tableau `Arr` et `a2` à pointer désormais sur une indirection, les valeurs 0 et 8 de l'élément d'indice 2 ayant été échangées entre le tableau `Arr` et l'indirection. Cette nouvelle situation est illustrée figure 6.

L'opération `reroot` a un coût proportionnel au nombre de pointeurs `Diff` qu'il faut suivre jusqu'au tableau `Arr`, mais elle ne sera faite que la première fois que l'on cherchera à accéder à une version antérieure d'un tableau persistant. Tout nouvel accès à ce tableau se fera alors de nouveau en temps constant. Autrement dit, on ne paye qu'une seule fois le prix du retour en arrière. Dans le contexte d'un algorithme effectuant du *backtrack* cette solution s'avère excellente. Si le nombre d'opérations sur les tableaux est bien supérieur au nombre de retours en arrière, alors la complexité amortie des opérations `get` et `set` sera $O(1)$ en temps et en espace.

Il est important de noter que si en revanche on manipule simultanément plusieurs versions d'un même tableau persistant, ou alternativement avec de nombreux allers-retours, alors les performances seront moins bonnes car la fonction `reroot` passera son temps à retourner des chaînes de pointeurs.

Enfin, on note que la fonction `reroot` n'est pas récursive terminale. Ceci peut être problématique lorsque l'on manipule des tableaux persistants ayant subi de très nombreuses modifications. On peut facilement y remédier en effectuant une transformation CPS, sans vraiment dégrader les performances. Les tests que nous présentons plus loin sont d'ailleurs effectuées sur la version CPS.

2.4. Deux dernières améliorations

On peut encore améliorer cette solution. La première idée consiste à ne rien faire du tout lors de l'opération set(t, i, v) lorsque i est déjà associé à v dans t. On évite ainsi la création d'une indirection inutile, et donc l'allocation de deux blocs mémoire. Cela est particulièrement efficace dans le contexte de la structure *union-find*, car la compression de chemin amène rapidement les éléments à pointer directement sur leur représentant et on évite alors le coût d'une opération `set` en fait inutile. (On aurait pu de manière équivalente dérouler une fois la fonction `find_aux` et traiter le cas particulier d'un chemin de longueur 1.)

La seconde idée consiste à remarquer que dans un contexte où l'on effectue *uniquement du backtracking*, alors il est inutile de maintenir le contenu des tableaux persistants qui deviennent inaccessibles au moment où l'on revient en arrière. De toutes façons, ces valeurs vont être immédiatement ramassées par le GC. On peut donc améliorer encore les performances des tableaux persistants dans le cas très particulier de notre utilisation, c'est-à-dire où l'on revient en arrière sur un tableau persistant t sans garder de pointeurs sur les versions ultérieures de t.

La première modification consiste à introduire une valeur `Invalid` dénotant un tableau persistant auquel il n'est plus possible d'accéder :

```
type t = data ref
and data =
  | Arr of int array
  | Diff of int × int × t
  | Invalid
```

Ensuite, il suffit de modifier la fonction `reroot` de manière à ne pas retourner la chaîne de pointeurs menant au tableau `Arr`, mais simplement à modifier le contenu de ce tableau :

```
let rec reroot t = match !t with
  | Arr _ → ()
  | Diff (i, v, t') →
      reroot t';
      begin match !t' with
        | Arr a as n →
            a.(i) ← v;
            t := n;
            t' := Invalid
        | Diff _ | Invalid → assert false
      end
  | Invalid → assert false
```

On économise en particulier l'allocation d'un constructeur `Diff`. Le reste du code est inchangé mais échoue si on cherche à accéder à un tableau persistant invalide.

Il est important de noter que la structure obtenue n'est donc que *semi-persistante* puisque les allers-retours sont maintenant interdits (à la différence des précédentes solutions qui étaient toutes persistantes).

Il est frappant de constater que la structure de données à laquelle nous aboutissons n'est rien d'autre qu'un tableau accompagné d'une pile d'opérations qu'il faudra appliquer pour revenir à un état précédent, c'est-à-dire exactement l'idiome de programmation impérative consistant à maintenir une pile de *undo*s. Mais à la différence d'une programmation impérative où cette pile est manipulée explicitement et à l'extérieur de la structure de données, elle est ici entièrement cachée à l'intérieur d'un module qui donne l'*illusion* de la persistance.

3. Performances

Nous avons cherché à tester les performances de notre solution dans une situation aussi réaliste que possible. Pour cela, nous avons examiné l'utilisation faite par une procédure de décision [6] de la structure *union-find* sous-jacente. On distingue trois paramètres :

- le nombre de retour en arrière ;
- le nombre total d'opérations `find` et `union` entre deux branchements, noté N ;
- la proportion d'opérations `union` par rapport aux opérations `find`, noté p.

Sur les tests que nous avons effectués avec la procédure de décision, il s'avère d'une part que le nombre de retour en arrière est relativement faible, et d'autre part que la proportion d'opérations `union` est également assez faible. Nous avons donc choisi d'effectuer nos tests en suivant les mêmes branchements que dans le parcours d'un arbre binaire complet de hauteur 4. Entre chaque nœud et ses deux fils, on effectue exactement N opérations. Ce parcours est effectué pour $N = 20000$, $N = 100000$ et $N = 500000$, et des valeurs de p de 5, 10 et 15%.

Le tableau suivant présente les temps d'exécution obtenus[3] pour diverses solutions :

p	5%	5%	5%	10%	10%	10%	15%	15%	15%
N	20000	100000	500000	20000	100000	500000	20000	100000	500000
Tarjan	0.31	2.23	12.50	0.33	2.34	12.90	0.34	2.36	13.20
2.3	0.52	3.03	17.10	0.81	4.78	26.80	1.16	6.78	37.90
2.4a	0.36	2.08	12.30	0.46	2.76	15.90	0.64	3.58	20.70
2.4b	0.34	2.01	11.70	0.42	2.54	14.90	0.52	3.21	18.70
defun.	**0.33**	**1.90**	**11.30**	**0.41**	**2.45**	**14.40**	**0.52**	**3.14**	**17.80**
naïve	0.76	5.28	37.50	1.22	9.14	63.80	40.40	>10mn	>10mn
maps	1.52	10.60	67.90	2.45	17.50	116.00	3.42	24.70	167.00

La première ligne donne les résultats de la version impérative de Tarjan. Bien entendu, celle-ci est utilisée ici incorrectement mais sert uniquement de référence. Les quatre lignes suivantes sont les raffinements successifs de notre solution : celle de la section 2.3, puis les deux améliorations de la section 2.4, puis enfin la version finale *défonctorisée* manuellement. Les deux dernières lignes correspondent à des solutions purement applicatives, à titre de comparaison : la première (naïve) est un AVL pour les liaisons sans compression de chemin ni niveaux, et la seconde est l'application du foncteur de la section 2.1 à des tableaux persistants réalisés par des AVLs.

4. Preuve formelle de correction

Même si la solution proposée est conceptuellement simple, les structures de données utilisées sont relativement complexes de part la présence de nombreux effets de bord, tant dans les tableaux persistants que dans la version persistante de l'algorithme de Tarjan. Pour cette raison, il nous a paru nécessaire de faire la preuve formelle de la correction de ces deux structures de données. Nous détaillons ici les grandes lignes de cette preuve, menée dans l'assistant de preuve Coq [1].

On suppose ici une certaine familiarité avec le système Coq. L'intégralité de ce développement est disponible en ligne[4]. La section 4.1 présente la formalisation des tableaux persistants puis la section 4.2 celle de la structure *union-find* persistante présentée dans la section 2.3.

[3]Les temps sont mesurés en secondes et correspondent à du temps CPU sur un Pentium IV 2,4 GHz.

[4]`http://www.lri.fr/~filliatr/puf/`

4.1. Tableaux persistants

Pour rendre compte des effets de bord et notamment de la possibilité d'alias dans le tas OCAML (les tableaux persistants sont des références, vers des structures contenant elles-mêmes d'autres références, et elles peuvent être partagées entre plusieurs versions d'un tableau persistant), il convient de modéliser le tas d'une manière relativement bas niveau. Mais il n'est cependant pas nécessaire de descendre jusqu'à un niveau de détail trop important, tel qu'expliciter l'organisation des blocs dans le tas, leur taille, etc.

Pour modéliser les références, on introduit un type abstrait `pointer` représentant la valeur d'une référence. Le tas est alors vu comme un dictionnaire associant à chaque référence la valeur qu'elle désigne. Ce dictionnaire est simplement axiomatisé, sous la forme d'un module `PM` déclarant un type abstrait polymorphe `t`, une fonction `find` retournant la valeur éventuellement associée à une référence (soit `None` si aucune valeur n'est associée, soit `Some` v sinon), une fonction `add` pour ajouter une nouvelle association, et enfin une fonction `new` pour obtenir une nouvelle référence (*i.e.* qui n'est pas encore associée à une valeur). Trois axiomes décrivent tout ce qu'il y a à savoir sur ces trois fonctions. La déclaration de ce module est donnée en annexe.

Le type des tableaux persistants est directement modélisé par le type `pointer` des références. Quant au type `data`, il reste un type somme avec deux constructeurs `Arr` et `Diff` :

```
Inductive data : Set :=
  | Arr : data
  | Diff  : Z → Z → pointer → data.
```

On utilise ici le fait que le tableau en argument du constructeur `Arr` est unique pour ne pas le représenter dans le type `data`. On modélise alors le tas OCAML par une paire contenant d'une part le dictionnaire associant les références aux valeurs de type `data`, et d'autre part le contenu du tableau usuel désigné par `Arr`, directement sous la forme d'une fonction de `Z` dans `Z` :

```
Record mem : Set := { ref : PM.t data; arr : Z→Z }.
```

Il est clair que l'on ne modélise ici que la partie du tas relative à un seul tableau persistant et ses descendants, mais cela est suffisant pour cette preuve de correction (il n'y a pas d'opération prenant en argument deux tableaux persistants et qui nécessiterait donc une modélisation plus complexe).

Ainsi définie, la modélisation inclut beaucoup plus de situations possibles dans le tas que ce que les seules opérations `create` et `set` permettent d'obtenir (de la même façon que la seule donnée des types OCAML n'exclut pas *a priori* de construire par exemple des valeurs cycliques). Pour rendre compte de l'invariant de la structure de tableau persistant, on introduit le prédicat inductif suivant, `pa_valid`, qui indique qu'une référence désigne un tableau persistant bien formé :

```
Inductive pa_valid (m: mem) : pointer → Prop :=
  | array_pa_valid :
     ∀p, PM.find (ref m) p = Some Arr → pa_valid m p
  | diff_pa_valid :
     ∀p i v p', PM.find (ref m) p = Some (Diff i v p') →
     pa_valid m p' → pa_valid m p.
```

Cette définition indique que la référence pointe soit sur une valeur de la forme `Arr`, soit sur une valeur de la forme `Diff` i v p' avec p' désignant lui-même un tableau persistant bien formé. Par sa nature inductive, elle implique également l'absence de cycle.

Enfin, pour exprimer les spécifications des fonctions `get` et `set`, on va relier chaque tableau persistant à la fonction sur $\{0, 1, \ldots, n-1\}$ qu'il représente. On représente cette fonction simplement par une valeur de type `Z→Z`, et on définit donc le prédicat inductif suivant reliant une référence à une fonction :

```
Inductive pa_model (m: mem) : pointer → (Z → Z) → Prop :=
  | pa_model_array :
     ∀p, PM.find (ref m) p = Some Arr → pa_model m p (arr m)
  | pa_model_diff :
     ∀p i v p', PM.find (ref m) p = Some (Diff i v p') →
     ∀f, pa_model m p' f → pa_model m p (upd f i v).
```

où la fonction `upd` est la mise à jour d'une fonction en un point, à savoir :

```
Definition upd (f:Z→Z) (i:Z) (v:Z) :=
  fun j ⇒ if Z_eq_dec j i then v else f j.
```

Comme on le constate, la définition de `pa_model` est analogue à celle de `pa_valid`. Il est même clair que l'on a `pa_valid` m p dès lors que l'on a `pa_model` m p f pour une certaine fonction f. Mais il est préférable de distinguer les deux, comme nous allons le montrer tout de suite.

Nous pouvons à présent spécifier les fonctions `get` et `set`. En optant pour des fonctions fortement spécifiées (*i.e.* contenant leur spécification dans leur type, par le biais de types dépendants), une spécification de `get` peut être :

```
Definition get :
  ∀m, ∀p, pa_valid m p →
  ∀i, { v:Z | ∀f, pa_model m p f → v = f i }.
```

On exprime en précondition que la référence p doit désigner un tableau persistant valide, et en postcondition que la valeur renvoyée v doit être f v pour tout fonction f que modélise p à travers la mémoire m. La preuve de `get` ne semble pas difficile, tant son énoncé paraît être tautologique. Elle soulève cependant une problème de taille : celui de la terminaison. En effet, la fonction `get` ne termine uniquement que parce qu'on a supposé que la référence p désigne un tableau persistant valide. Dans le cas contraire, il pourrait y avoir une circularité dans le tas (telle que celle obtenue en écrivant `let rec p = ref (Diff (0,0,p))`) et dès lors des appels à `get` ne terminant pas. Si la fonction `get` termine, c'est parce que la référence p mène en un nombre fini d'étapes à la valeur `Arr`. Pour traduire cette propriété, on introduit le prédicat `dist` m p n traduisant que la distance de p à la valeur `Arr` est l'entier naturel n. On peut alors introduire la relation R m, pour une mémoire m, indiquant qu'un pointeur est plus proche qu'un autre de la valeur `Arr` :

$$R\ m\ p_1\ p_2 \equiv \exists n_1,\ \exists n_2,\ \texttt{dist}\ m\ p_1\ n_1 \wedge \texttt{dist}\ m\ p_2\ n_2 \wedge n_1 < n_2$$

On peut alors définir la fonction `get` par récurrence bien fondée sur cette relation. Comme il est facile de montrer que si p pointe sur la valeur `Diff` i v p' alors R m p' p, sous l'hypothèse `pa_valid` m p, la terminaison de `get` en découle[5].

La fonction `set` a une spécification un peu plus complexe car elle doit évidemment traduire l'effet de l'affectation mais également la persistance du tableau passé en argument. Une spécification possible est la suivante :

```
Definition set :
  ∀m:mem, ∀p:pointer, ∀i v:Z,
  pa_valid m p →
  { p':pointer & { m':mem |
    ∀f, pa_model m p f →
       pa_model m' p f ∧ pa_model m' p' (upd f i v) } }.
```

[5]On aurait pu également procéder par récurrence structurelle sur la *preuve* de `pa_valid`, ainsi qu'expliqué dans le *Coq'Art* [4] section 15.4.

Ici la fonction renvoie non seulement le résultat p' mais également le nouvel état mémoire m'. La précondition reste la même, à savoir la validité du tableau persistant passé en argument. La postcondition, quant à elle, indique pour toute fonction f que modélisait p dans la mémoire initiale m, alors p continue de modéliser cette même fonction f dans la nouvelle mémoire m' et p' modélise le résultat de l'affectation, à savoir la fonction `upd` f i v.

La preuve de `set` ne contient qu'une seule difficulté : il faut exprimer que l'allocation d'une nouvelle référence ne modifie pas la structure des tableaux persistants déjà alloués. Ceci se traduit par le lemme de séparation suivant, où la nouvelle référence est obtenue par la fonction `PM.new` et initialisée avec une valeur d quelconque :

```
Lemma pa_model_sep :
  ∀m, ∀p, ∀d, ∀f,
  pa_model m p f→
  pa_model (Build_mem (PM.add (ref m) (PM.new (ref m)) d) (arr m)) p f.
```

4.2. Union-find persistant

Pour modéliser la structure *union-find*, on se fixe une fois pour toutes le nombre d'éléments, sous la forme d'un paramètre `N` de type `Z`. Sans perte de généralité, on peut omettre la gestion des niveaux, qui n'entre pas en compte dans la correction, mais seulement dans la complexité. La structure *union-find* est donc réduite à un tableau persistant ; on peut donc conserver la modélisation du tas de la section précédente et représenter directement une structure *union-find* par une valeur du type `pointer`.

Comme pour les tableaux persistants, il faut commencer par définir une notion de validité. En effet, une structure *union-find* n'est pas constituée d'un tableau persistant quelconque, mais d'un tableau persistant représentant une fonction avec la propriété bien particulière d'associer un représentant à chaque entier de $[0, N-1]$, en une ou plusieurs étapes. Comme dans le cas des tableaux persistants, cette propriété de validité est essentielle pour garantir la terminaison. Pour la définir, on commence par définir la notion de représentant d'une classe, pour une modélisation du tableau des liaisons *i.e.* une fonction f de `Z` dans `Z` :

```
Inductive repr (f: Z→Z) : Z→Z→Prop :=
  | repr_zero :
      ∀i, f i = i → repr f i i
  | repr_succ :
      ∀i j r, f i = j → 0≤j<N → ~j=i → repr f j r → repr f i r.
```

On peut alors définir la notion de validité comme la validité du tableau persistant d'une part, et l'existence d'un représentant pour tout entier de $[0, N-1]$ d'autre part :

```
Definition reprf (f:Z→Z) :=
  (∀i, 0≤i<N → 0≤f i<N) ∧
  (∀i, 0≤i<N → ∃j, repr f i j).
Definition uf_valid (m:mem) (p:pointer) :=
  pa_valid m p ∧
  ∀f, pa_model m p f → reprf f.
```

Spécifier la fonction `find` ne consiste pas uniquement à dire qu'elle renvoie une valeur étant effectivement le représentant. En effet, du fait de la compression de chemins, la fonction `find` modifie l'état mémoire, et il faut donc spécifier également l'invariance des classes par cette compression. On commence donc par définir la propriété pour deux fonctions de définir le même ensemble de représentants :

```
Definition same_reprs f₁ f₂ :=
  ∀i, 0≤i<N → ∀j, repr f₁ i j ↔ repr f₂ i j.
```

On peut alors spécifier ainsi la fonction `find` (en fait la fonction `find_aux` *i.e.* celle qui retourne le nouveau tableau persistant contenant la compression de chemins, modélisé ici sous la forme du pointeur p') :

```
Definition find :
  ∀m, ∀p, uf_valid m p →
  ∀x, 0≤x<N →
  { r:Z & { p':pointer & { m':mem |
    uf_valid m' p' ∧
    ∀f, pa_model m p f → repr f x r ∧
    ∀f', pa_model m' p' f' → same_reprs f f' } } }.
```

Là encore, la preuve nécessite une récurrence bien fondée, cette fois sur la distance séparant x de son représentant.

Pour spécifier la fonction `union`, on a besoin de la notion d'éléments équivalents *i.e.* appartenant à la même classe :

```
Definition equiv f x y :=
  ∀rx ry, repr f x rx → repr f y ry → rx=ry.
```

On peut alors spécifier ainsi la fonction `union` (l'affectation `h.father <- f` est modélisée ici par le pointeur p_1 renvoyé) :

```
Definition union :
  ∀m, ∀p, uf_valid m p →
  ∀x y, 0≤x<N → 0≤y<N →
  { p':pointer & { p₁:pointer & { m':mem |
       uf_valid m' p₁ ∧ uf_valid m' p'  ∧
       ∀f₁, pa_model m p f₁ →
       ((∀f₂, pa_model m' p₁ f₂ → same_reprs f₁ f₂) ∧
        (∀f', pa_model m' p' f' →
         ∀a b, 0≤a<N → 0≤b<N →
            (equiv f' a b ↔
             (equiv f₁ a b ∨ ((equiv f₁ a x ∧ equiv f₁ b y) ∨
                              (equiv f₁ b x ∧ equiv f₁ a y)))))) }}}.
```

La postcondition relativement complexe traduit plusieurs propriétés : d'une part la persistance de la structure initiale (elle est toujours valide dans la nouvelle mémoire et définit le même ensemble de représentants) ; d'autre part la validité de la nouvelle structure ; enfin, la propriété fonctionnelle proprement dite, à savoir que `union` réalise bien l'union des classes repsectives de x et y. On exprime ceci en disant que pour tous éléments a et b, a et b sont dans la même classe à l'issue de l'union (fonction f') si et seulement si ils l'étaient déjà auparavant (fonction f_1) ou bien a et b étaient tous deux des éléments des classes initiales de x et y.

Au total, la formalisation des tableaux persistants représente 300 lignes de tactiques Coq et celle de la structure *union-find* 700 lignes, en incluant à chaque fois tous les lemmes et définitions auxiliaires.

5. Conclusion

Nous avons présenté une structure de données persistante pour le problème *union-find* dont les performances sont comparables à celle de la version impérative de Tarjan [13]. Cette solution est construite d'une part à partir d'une version persistante de l'algorithme de Tarjan, paramétrée par une structure de tableaux persistants, et d'autre part à l'aide d'une réalisation efficace de tableaux persistants en suivant une idée de Baker [9, 3]. Bien que persistantes, ces deux parties de la solution font un usage massif d'effets de bord. Contrairement aux idées reçues, la persistance n'est pas synonyme de programmation purement applicative (même si d'excellents ouvrages tels que le livre d'Okasaki [11] se contentent de solutions essentiellement applicatives). En contrepartie, il est moins facile de se convaincre de la correction de la solution obtenue, et notamment de son caractère persistant, et c'est pourquoi nous avons présenté une preuve formelle de correction de ce code.

La version la plus efficace à laquelle nous sommes parvenus utilise en fait des tableaux qui ne sont pas complètement persistants. En effet, il n'est correct de les utiliser que pour revenir à des versions antérieures (ce qui est l'utilisation typique d'une structure persistante dans un contexte de *backtracking*). Cette *semi-persistance* a pour l'instant un caractère dynamique (la donnée est invalidée lorsque l'on revient en arrière) mais il serait plus efficace encore de déterminer statiquement l'utilisation licite de cette semi-persistance.

Remerciements. Nous remercions sincèrement les membres du projet Proval pour toutes les discussions que nous avons pu avoir autour du problème de l'*union-find* persistant depuis longtemps. Nous remercions en particulier Claude Marché[6] pour nous avoir indiqué l'article de T.-R. Chuang [5] et Christine Paulin pour nous avoir incités à effectuer directement une preuve avec Coq sans passer par un outil de preuve de programmes C, comme nous l'avions imaginé initialement.

Références

[1] L'assistant de preuve Coq. `http://coq.inria.fr/`.

[2] Le langage Objective Caml. `http://caml.inria.fr/`.

[3] Henry G. Baker. Shallow binding makes functional arrays fast. *SIGPLAN Not.*, 26(8) :145–147, 1991.

[4] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development.* Texts in Theoretical Computer Science. An EATCS Series. Springer Verlag, 2004. `http://www.labri.fr/Perso/~casteran/CoqArt/index.html`.

[5] Tyng-Ruey Chuang. Fully persistent arrays for efficient incremental updates and voluminous reads. In *ESOP'92 : Symposium proceedings on 4th European symposium on programming*, pages 110–129, London, UK, 1992. Springer-Verlag.

[6] Sylvain Conchon and Evelyne Contejean. Ergo : A Decision Procedure for Program Verification. `http://ergo.lri.fr/`.

[7] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press/McGraw-Hill, 1990.

[8] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38(1) :86–124, 1989.

[9] Jr. Henry G. Baker. Shallow binding in Lisp 1.5. *Commun. ACM*, 21(7) :565–569, 1978.

[10] G. Nelson and D. C. Oppen. Fast decision procedures based on congruence closure. *Journal of the ACM*, 27 :356–364, 1980.

[11] Chris Okasaki. *Purely Functional Data Structures.* Cambridge University Press, 1998.

[12] R. E. Shostak. Deciding combinations of theories. *Journal of the ACM*, 31 :1–12, 1984.

[13] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2) :215–225, 1975.

[6]Nous remercions également Claude pour son slogan « le beurre et le Tarjan du beurre ».

A. Algorithme de Tarjan

```
type t = { rank : int array; father : int array }

let create n =
  { rank = Array.create n 0;
    father = Array.init n (fun i → i) }

let rec find_aux f i =
  if f.(i) == i then
    i
  else
    let ri = find_aux f f.(i) in
    f.(i) ← ri;
    ri

let find h x = find_aux h.father x

let union {rank=r; father=f} x y =
  let rx = find_aux f x in
  let ry = find_aux f y in
  if rx != ry then begin
    if r.(rx) > r.(ry) then
      f.(ry) ← rx
    else if r.(rx) < r.(ry) then
      f.(rx) ← ry
    else begin
      r.(rx) ← r.(rx) + 1;
      f.(ry) ← rx
    end
  end
```

B. Une modélisation simple des références en Coq

```
Module PM.
  Parameter t : Set → Set.
  Parameter find : ∀a, t a → pointer → option a.
  Parameter add : ∀a, t a → pointer → a → t a.
  Parameter new : ∀a, t a → pointer.
  Axiom find_add_eq :
    ∀a, ∀m:t a, ∀p:pointer, ∀v:a,
    find (add m p v) p = Some v.
  Axiom find_add_neq :
    ∀a, ∀m:t a, ∀p p':pointer, ∀v:a,
    ~p'=p → find (add m p v) p' = find m p'.
  Axiom find_new :
    ∀a, ∀m:t a, find m (new m) = None.
End PM.
```